Self-stabilizing robot formations over unreliable networks

pdfauthor

Self-Stabilizing Robot Formations over Unreliable Networks

Seth Gilbert, Ecole Polytechnique Fédérale, Lausanne
Nancy Lynch, Massachusetts Institute of Technology
Sayan Mitra, University of Illinois at Urbana-Champaign
and
Tina Nolte, Massachusetts Institute of Technology

We describe how a set of mobile robots can arrange themselves on any specified curve on the plane in the presence of dynamic changes both in the underlying ad hoc network and the set of participating robots. Our strategy is for the mobile robots to implement a *self-stabilizing virtual layer* consisting of mobile client nodes, stationary Virtual Nodes (VNs), and local broadcast communication. The VNs are associated with predetermined regions in the plane and coordinate among themselves to distribute the client nodes relatively uniformly among the VNs' regions. Each VN directs its local client nodes to align themselves on the local portion of the target curve. The resulting motion coordination protocol is self-stabilizing, in that each robot can begin the execution in any arbitrary state and at any arbitrary location in the plane. In addition, self-stabilization ensures that the robots can adapt to changes in the desired target formation.

Categories and Subject Descriptors: F.1.2 [**Computation by Abstract Devices**]: Modes of Computation–*Interactive and Reactive Computation*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: cooperative mobile robotics, distributed algorithms, pattern formation, self-stabilization

1. INTRODUCTION

In this paper, we study the problem of coordinating the behavior of autonomous mobile robots, even as robots join and leave the system. Consider, for example, a system of firefighting robots deployed throughout forests and other arid wilderness areas. Significant levels of coordination are required in order to combat the fire: to prevent the fire from spreading, it has to be surrounded; to put out the fire, firefighters need to create "firebreaks" and spray water; they need to direct the actions of (potentially autonomous) helicopters carrying water. All this has to be achieved while the set of participating agents is changing and despite unreliable (often, wireless) communication between agents. Similar scenarios arise in a variety of contexts, including search and rescue, emergency disaster response, remote surveillance, and military engagement, among many others. In fact, autonomous coordination has long been a central problem in mobile robotics.

S. Gilbert, Laboratory for Distributed Programming, Ecole Polytechnique Fédérale, Lausanne. Email: seth.gilbert@epfl.ch. N. Lynch, Dept. of Electrical Engineering and Computer Science, MIT Email:lynch@csail.mit.edu. S. Mitra, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Supported by NSF CSR program (Embedded & Hybrid systems area) under grant NSF CNS-0614993. Email:mitras@crhc.uiuc.edu. T. Nolte, Dept. of Electrical Engineering and Computer Science, MIT tnolte@mit.edu.

We focus on a generic coordination problem that, we believe, captures many of the complexities associated with coordination in real-world scenarios. We assume that the mobile robots are deployed in a large two-dimensional plane, and that they can coordinate their actions by local communication using wireless radios. The robots must arrange themselves to form a particular pattern, specifically, a continuous curve drawn in the plane. The robots must spread themselves uniformly along this curve. In the firefighting example described above, this curve might form the perimeter of the fire.

These types of coordination problems can be quite challenging due to the dynamic and unpredictable environment that is inherent to wireless ad hoc networks. Robots may be continuously joining and leaving the system, and they may fail unpredictably. In addition, wireless communication is notoriously unreliable due to collisions, contention, and various wireless interference.

Virtual Infrastructure. Recently, *virtual infrastructure* has been proposed as a new tool for building reliable and robust applications in unreliable and unpredictable wireless ad hoc networks (e.g., [Dolev et al. 2003; Dolev et al. 2005; Chockler et al. 2008]). The basic principle motivating virtual infrastructure is that many of the challenges resulting from dynamic networks could be obviated if there were reliable network infrastructures available. We believe that coordinating mobile robots is exactly one of those problems. Unfortunately, in many contexts, such infrastructure is unavailable. The virtual infrastructure abstraction emulates real reliable infrastructure in ad hoc networks. Thus, it provides a programming abstraction that *assumes* reliable infrastructure, and thus simplifies the problem of developing applications. It has already been observed that virtual infrastructure simplifies several problems in wireless ad hoc networks, including distributed shared memory implementations [Dolev et al. 2003], tracking mobile devices [Nolte and Lynch 2007b], geographic routing [Dolev et al. 2005b], and point-to-point routing [Dolev et al. 2004].

In this paper, we rely on a virtual infrastructure known as the Virtual Stationary Automata Layer (VSA Layer) [Dolev et al. 2005a; Nolte and Lynch 2007a]. In the VSA Layer, each robot is modeled as a *client*; clients interact with *virtual stationary automata* (VSAs) via a (virtual) communication service. VSAs are distributed throughout the world, each assigned to its own unique region. VSAs remain always at a known and predictable location, and they are less likely to fail than any individual mobile robot. Notice that the VSAs do not actually exist in the real world; they are emulated by the underlying mobile robots. It is for this reason that a VSA is more reliable: it is as reliable as the entire collection of mobile nodes that are participating in its emulation.

In fact, we believe that the VSA Layer is particularly suitable for solving problems such as motion coordination due to the failure properties of VSAs. In many ways, VSAs and mobile robots have a *fate sharing* relationship: a VSA responsible for some specific region fails only when all the robots in its region fail or leave. Thus, as long as there are robots to coordination, the VSA is guaranteed to remain alive. Conversely, whenever the VSA is failed, there are no robots alive or nearby that rely on the VSA. Thus from the perspective of the mobile robots, the VSAs appear completely reliable.

We do not address here the problem of implementing virtual infrastructure; instead, we refer the reader to [Dolev et al. 2005a; Nolte and Lynch 2007a; Nolte 2008]. In these papers, we show how to emulate VSAs using mobile, wireless devices much like the mobile robots discussed in this paper. The wireless devices have access to a synchronized time service, and a localization service (such as a GPS device), and they communicate via reliable and timely radio broadcasts. In Section 4.2 we provide a brief overview of these requirements, and of the protocol for implementing virtual infrastructure.

Coordinating Mobile Robots. Our main contribution is a technique for using the VSA Layer to implement a reliable and robust protocol for coordinating mobile robots. The key simplifying fact of the VSA Layer is that reliable VSAs are distributed evenly throughout the world. Thus, each VSA is responsible for organizing the mobile robots in some region of the world. As the execution progresses, each VSA directs the robots in its region as to how they should move. There are two further technical issues that arise in our protocol: how does the VSA collect the information that it needs (and how much information does it need), and how, given only limited information, does the VSA direct the mobile robots in order to ensure a good distribution of robots along the curve. In brief, we address these issues as follows.

In order to determine where each robot should go, the VSA needs to collect information about the current distribution of the robots. Each robot in a region notifies the responsible VSA of its presence, and, in addition, the VSAs exchange information with their neighboring VSAs. Thus, each VSA maintains a local view of the number of robots it and its neighbors are responsible for. Of note, the VSA only collects *local* information, i.e., the distribution of robots in its own and neighboring regions. It does not, for example, collect information about the location of every robot, as this would take prohibitively long (and might not even be possible, if the network induced by the robots is initially partitioned).

Using this local information about robot distribution, the VSA decides how many robots to keep in its own region, and how many to distribute to its neighbors. By carefully re-allocating robots, the VSAs cause the robots to diffuse throughout the network. The diffusion process is biased by the length of the curve in each region to ensure that the concentration of robots reflects the needs of each VSA. Once the robots have diffused throughout the network, each VSA assigns the robots to locations on the curve.

Self-Stabilization. In order that the robot coordination be truly robust, our coordination protocol is *self-stabilizing*. In general, a self-stabilizing system is one which regains normal functionality and behavior sometime after disturbances, such as node failures and message losses cease.

In our case, this means that each robot can begin in an arbitrary state, in an arbitrary location in the network, and with an arbitrary view of the world. Even so, the distribution of the robots will still converge to the specified curve. When combined with a self-stabilizing implementation of the VSA Layer, as is presented in [Dolev et al. 2005a; Nolte and Lynch 2007a], we end up with entirely self-stabilizing solution for the problem of autonomous robot coordination.

We believe that self-stabilization is particularly important in the context of wireless networks. Most of the time, wireless communication works reasonably well. Most of the time, mobile robots act as expected. Our algorithms (and those for implementing virtual infrastructure) rely on this common case behavior: communication is reliable, mobile robots move as directed, GPS devices return correct locations, etc.

And yet, in the real world, mobile robots are not perfectly reliable. Sometimes, they fail to move exactly as expected, due to minor errors in actuating their motors, bad sensor readings, or perhaps, due to incorrectly detecting (or not detecting) obstacles that must be avoided[1] Sometimes GPS devices return an incorrect location, or cannot acquire a sufficient number of satellites to return good localization information. Sometimes there is electromagnetic interference that disrupts communication. Sometimes wireless messages are lost due to too much contention, i.e., too many different applications attempting to communicate on a limited bandwidth. There are a wide variety of problems that can occur in a deployment of mobile robots, and a wide variety of disruptions that can interfere with wireless communication. All of these challenges result in deviations from the common case setting for which our algorithms are designed.

Thus, there are clearly two options available for coping with this situation. One option is to design algorithms that can directly handle these challenges, and yet still achieve the desired outcome. Pursuing this direction leads to algorithms that are immensely complicated. Moreover, it requires carefully enumerating all the possible problems that might occur; any unexpected disruption can lead to a complete failure.

A second option, and the one that we pursue, is self-stabilization. A self-stabilizing algorithm can recover from all types of problems, as long as the errors are temporary. In effect, a self-stabilizing algorithm requires only that the robots and the wireless communication work correctly *most of the time*. In this way, such protocols are a classic example of the paradigm, *plan for the worst, expect the best.* Despite occasional problems, the mobile robots will converge to the desired formation.

Another advantage to self-stabilization is the capacity to cope with more dynamic coordination problems. In real-life scenarios, the required formation of the mobile nodes may change. In the firefighting example above, as the fire advances or retreats, the formation of firefighting robots must adapt. A self-stabilizing algorithm can adapt to these changes, continually re-arranging the robots along the newly chosen curve.

Proof Techniques. Analyzing self-stabilizing algorithm can be quite difficult, however, and another technical contribution of this paper is the exemplification of a proof technique for showing self-stabilization of systems implemented using virtual infrastructure. The proof technique has three parts. First, using invariant assertions and standard control theory results we show that from any initial state, the

[1] In fact, much of the prior work on fault-tolerant motion coordination has focused on the problems that arise when the robots have faulty vision or a bad sense of direction. Unlike these prior papers, we allow the robots to coordinate via radio broadcasts. Thus, problems caused by faulty vision are limited to disrupting the expected movement of the robots.

application protocol, in this case, the motion coordination algorithm converges to an *acceptable state*. Next, we show that the algorithm always reaches a *legal state* even when it starts from some arbitrary state after failures. From any legal state the algorithm gets to an acceptable state provided there are no further failures. Finally, using a simulation relation we show that the above set of legal states is in fact equal to the set of reachable states of the complete system—the coordination algorithm composed with the VSA layer. It has already been shown in [Dolev et al. 2005a; Nolte and Lynch 2007a] that the VSA layer itself is self-stabilizing. Thus, combining the stabilization of the VSA layer and the application protocol, we are able to conclude self-stabilization of the complete system.

Roadmap. The remainder of this paper is organized as follows:

- In Section 2, we discuss some of the related work.
- In Section 3, we introduce the underlying mathematical model used for specifying the VSA layer. We define Timed I/O Automata (TIOA), and a formalism for discussing their behaviors (i.e., traces and executions). We discuss how to transform a TIOA designed for a reliable network into a TIOA that executes in an unreliable system, and we define what it means for a TIOA to be self-stabilizing.
- In Section 4 we discuss the VSA Layer model. We present the overall VSA Layer architecture, describing the behavior of each of the underlying components. We also briefly describe how to emulate the VSA Layer in a wireless network.
- In Section 5 we begin by formally describing the motion coordination problem. Of note, we define a parameterized curve Γ, and define the quantized length of a segment of Γ. We then describe algorithm that solves the problem of motion coordination. The algorithm is divided into two components. The first part (i.e., the client code) runs on the mobile robots and simply sends updates to the VSA, receives responses from the VSAs, and then moves as directed. The second part (i.e., the server code) runs on the VSAs, and is responsible for planning the motion of the mobile robots.
- In Section 6, we show that the algorithm is correct. For the purpose of this section, we assume that the system begins in a good state, i.e., that the state has not been corrupted by Byzantine failures. We show that eventually, the mobile robots are appropriate distributed through the world, and that they arrange themselves evenly along the curve.
- In Section 7, we show that the algorithm is self-stabilizing. We define two legal sets of states, and argue that the system converges to first one, and then the second of these legal sets. We then argue that this second set of legal states is a "reachable" state from an initial state of the system via a simulation relation.
- We conclude in 8.

2. RELATED WORK

In the distributed computing literature, the idea of self-stabilization has been proposed an important way of engineering fault tolerance in systems that are inherently unreliable [Dolev 2000]. The idea of self-stabilization has been widely employed for designing resilient distributed systems over unreliable communication and computing components (see [Herman 1996] for a comprehensive list of applications).

The problem of motion coordination has been studied in a variety of contexts, focusing on several different goals: flocking [Jadbabaie et al. 2003]; rendezvous [Ando et al. 1999; Lin et al. 2003; Martinez et al. 2005]; aggregation [Gazi and Passino 2003]; deployment and regional coverage [Cortes et al. 2004]. Control theory literature contains several algorithms for achieving spatial patterns [Fax and Murray 2004; Clavaski et al. 2003; Blondel et al. 2005; Olfati-Saber et al. 2007]. These algorithms assume that the agents process information and communicate synchronously, and hence, they are analyzed based on differential or difference equations models of the system. Convergence of this class of algorithms over unreliable and delay-prone communication channels have been studied recently in [Chandy et al. 2008].

Geometric pattern formation with vision-based models for mobile robots have been investigated in [Suzuki and Yamashita 1999; Prencipe 2001; Flocchini et al. 2001; Efrima and Peleg 2007; Prencipe 2000; Défago and Konagaya 2002]. In these weak models, the robots are oblivious, identical, anonymous, and often without memory of past actions. For the memoryless models, the algorithms for pattern formation are often automatically self-stabilizing. In [Défago and Konagaya 2002; Défago and Souissi 2008], for instance, a self-stabilizing algorithm for forming a circle has been presented. These weak models have been used for characterizing the class of patterns that can be formed and for studying the computational complexity of formation algorithms, under different assumptions about the level of common knowledge amongst agents, such as, knowledge of distance, direction, and coordinates [Suzuki and Yamashita 1999; Prencipe 2000].

We have previously presented a protocol for coordinating mobile devices using virtual infrastructure in [Lynch et al. 2005]. The paper described how to implement a simple asynchronous virtual infrastructure, and proposed a protocol for motion coordination. This earlier protocol relies on a weaker (i.e., untimed) virtual layer (see [Dolev et al. 2005a; Nolte and Lynch 2007a]), while the current relies on a stronger (i.e., timed) virtual layer. As a result, our new coordination protocol is somewhat simpler and more elegant than the previous version. Moreover, the new protocol is self-stabilizing, which allows both for better fault-tolerance, and also the ability to tolerate dynamic changes in the desired pattern of motion. Virtual infrastructure has also been considered in [Brown 2007] for collision prevention of airplanes.

3. PRELIMINARIES

In this paper we mathematically model the the Virtual Infrastructure and all components of our algorithms using the *Timed Input/Output Automata* (TIOA) framework. TIOA is a mathematical modeling framework for real-time, distributed systems that interact with the physical world. Here we present key concepts of the framework and refer the reader to [Kaynar et al. 2005] for further details.

3.1 Timed I/O Automata

A *Timed I/O Automaton* is a non-deterministic state transition system in which the state may change either (a) instantaneously, by means of a *discrete transition*, or (b) continuously, over an interval of time, by following a *trajectory*. Let V be a set of variables. Each variable $v \in V$ is associated with a *type* which defines the set of values v can take on. The set of valuations of V, that is, mappings from V to

values, is denoted by $val(V)$. Each variable may be *discrete* or *continuous*. Discrete variables are used to model protocol data structures, while continuous variables are used to model physical quantities such as time, position, and velocity.

The semi-infinite real line $\mathbb{R}_{\geq 0}$ is used to model time. A *trajectory* τ for a set V of variables maps a left-closed interval of $\mathbb{R}_{\geq 0}$ with left endpoint 0 to $val(V)$. It models evolution of values of the variables over a time interval. The domain of τ is denoted by $\tau.dom$. We define $\tau.fstate \triangleq \tau(0)$. A trajectory is *closed* if $\tau.dom = [0, t]$ for some $t \in \mathbb{R}_{\geq 0}$, in which case we define $\tau.ltime \triangleq t$ and $\tau.lstate \triangleq \tau(t)$.

Definition 3.1. *A TIOA $\mathcal{A} = (X, Q, \Theta, A, \mathcal{D}, \mathcal{T})$ consists of (a) A set X of* variables*. (b) A non-empty set $Q \subseteq val(X)$ of* states*. (c) A non-empty set $\Theta \subseteq Q$ of* start states*. (d) A set A of* actions *partitioned into* input, output *and* internal *actions I, O, and H, (e) A set $\mathcal{D} \subseteq Q \times A \times Q$ of* discrete transitions*. If $(\mathbf{x}, a, \mathbf{x}') \in \mathcal{D}$, we often write $vx \xrightarrow{a} \mathbf{x}'$. An action $a \in A$ is said to be* enabled *at $\mathbf{x}$ iff $\mathbf{x} \xrightarrow{a} \mathbf{x}'$ for some $\mathbf{x}'$. (f) A set $\mathcal{T}$ of trajectories for X that is closed under prefix, suffix and concatenation.*[2]

In addition, $\mathcal{A}$ must be input action and input trajectory enabled.[3] We assume in this paper that the values of discrete variables do not change during trajectories.

We denote the components $X, Q, \mathcal{D}, \ldots$ of a TIOA $\mathcal{A}$ by $X_{\mathcal{A}}, Q_{\mathcal{A}}, \mathcal{D}_{\mathcal{A}}, \ldots$, respectively. For TIOA $\mathcal{A}_1$, we denote the components by $X_1, Q_1, \mathcal{D}_1, \ldots$.

Executions. An execution of $\mathcal{A}$ records the valuations of all variables and the occurrences of all actions over a particular run. An *execution fragment* of $\mathcal{A}$ is finite or infinite sequence $\tau_0 a_1 \tau_1 a_2 \cdots$ such that for every i, $\tau_i.lstate \xrightarrow{a_{i+1}} \tau_{i+1}.fstate$. An execution fragment is an *execution* if $\tau_0.fstate \in \Theta$. The first state of α, which we refer to as $\alpha.fstate$, is $\tau_0(0)$, and for a closed α (i.e., one that is finite and whose last trajectory is closed), its last state, $\alpha.lstate$, is the last state of its last trajectory. The *limit time* of α, $\alpha.ltime$, is defined to be $\sum_i \tau_i.ltime$. A state $\mathbf{x}$ of $\mathcal{A}$ is said to be *reachable* if there exists a closed execution α of $\mathcal{A}$ such that $\alpha.lstate = \mathbf{x}$. The sets of executions and reachable states of $\mathcal{A}$ are denoted $\mathsf{Execs}_{\mathcal{A}}$ and $\mathsf{Reach}_{\mathcal{A}}$. The set of execution fragments of $\mathcal{A}$ starting in states in a nonempty set L is denoted by $\mathsf{Frags}_{\mathcal{A}}^{L}$

A nonempty set of states $L \subseteq Q_{\mathcal{A}}$ is said to be a *legal set* for $\mathcal{A}$ if it is closed under the transitions and closed trajectories of $\mathcal{A}$. That is, a legal set satisfies the following: (1) if $(\mathbf{x}, a, \mathbf{x}') \in \mathcal{D}_A$ and $\mathbf{x} \in L$, then $\mathbf{x}' \in L$, and (2) if $\tau \in \mathcal{T}_{\mathcal{A}}$, τ is closed, and $\tau.fstate \in L$ then $\tau.lstate \in L$.

Traces. Often we are interested in studying the externally visible behavior of a TIOA $\mathcal{A}$. We define the *trace* corresponding to a given execution α by removing all internal actions, and replacing each trajectory τ with a representation of the time that elapses in τ. Thus, the trace of an execution α, denoted by $trace(\alpha)$, has information about input/output actions and the duration of time that elapses between the occurrence of successive input/output actions. The set of traces of $\mathcal{A}$ is defined as $\mathsf{Traces}_{\mathcal{A}} \triangleq \{\beta \mid \exists \alpha \in \mathsf{Execs}_{\mathcal{A}}, trace(\alpha) = \beta\}$.

[2]See Chapters 3 and 4 of [Kaynar et al. 2005] for formal definitions of these closure properties.
[3]See Chapters 64 of [Kaynar et al. 2005].

Implementation. Our proof techniques often rely on showing that any behavior of a given TIOA $\mathcal{A}$ is externally indistinguishable from some behavior of another TIOA $\mathcal{B}$. This is formalized by the notion of implementation. Two TIOAs are said to be *comparable* if their external interfaces are identical, that is, they have the same input and output actions. Given two comparable TIOAs $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{A}$ is said to *implement* $\mathcal{B}$, if $\mathsf{Traces}_{\mathcal{A}} \subseteq \mathsf{Traces}_{\mathcal{B}}$. The standard technique for proving that $\mathcal{A}$ implements $\mathcal{B}$ is to define a *simulation relation* $\mathcal{R} \subseteq Q_{\mathcal{A}} \times Q_{\mathcal{B}}$ which satisfies the following: if $\mathbf{x}\mathcal{R}\mathbf{y}$, then every one-step move of $\mathcal{A}$ from a state $\mathbf{x}$ simulates some execution fragment of $\mathcal{B}$ starting from $\mathbf{y}$, in such a way that: (1) the corresponding final states are also related by $\mathcal{R}$, and (2) the traces of the moves are identical (see [Kaynar et al. 2005], Section 4.5, for the formal definition).

Composition. It is convenient to model a complex system, such as our VSA layer, as a collection of TIOAs running in parallel and interacting through input and output actions. A pair of TIOAs are said to be *compatible* if they do not share variables or output actions, and if no internal action of either is an action of the other. The *composition* of two compatible TIOAs $\mathcal{A}$ and $\mathcal{B}$ is another TIOA which is denoted by $\mathcal{A}\|\mathcal{B}$. Binary composition is easily extended to any finite number of automata.

3.2 Failure transform for TIOAs

In this paper, we will describe algorithms that are self-stabilizing even in the face of ongoing mobile robot failures and recoveries. In order to model failures and recoveries, we introduce a general *failure transformation* of TIOAs. Thus, we can define a TIOA $\mathcal{A}$ in terms of its *correct* behavior, and then analyze the behavior of $Fail(\mathcal{A})$, which models the behavior of $\mathcal{A}$ in a failure-prone system.

A TIOA $\mathcal{A}$ is said to be is *fail-transformable* if it does not have the variable $failed$, and it does not have actions fail or restart. If $\mathcal{A}$ is fail-transformable, then the transformed automaton $Fail(\mathcal{A})$ is constructed from $\mathcal{A}$ by adding the discrete state variable *failed*, a Boolean that indicates whether or not the machine is failed, and two additional input actions, fail and restart. The states of $Fail(\mathcal{A})$ are the states of $\mathcal{A}$, together with a valuation of *failed*. The start states $Fail(\mathcal{A})$ are the states in which *failed* is arbitrary, but if it is false, then the rest of the variables are set to values consistent with a start state of $\mathcal{A}$. The discrete transitions of $Fail(\mathcal{A})$ are derived from those of $\mathcal{A}$ as follows: (1) an ordinary input transition at a failed state leaves the state unchanged, (2) an ordinary input transition at a non-failed state is the same as in $\mathcal{A}$, (3) a fail action sets $failed$ to true, (4) if a restart action occurs at a failed state then $failed$ is set to false and the other state variables are set to a start state of $\mathcal{A}$; otherwise, it does not change the state.

The set of trajectories of $Fail(\mathcal{A})$ is the union of two disjoint subsets, one for each value of the *failed* variable. The subset for $failed =$ false consists of trajectories of $\mathcal{A}$ with the addition of the constant value for $failed$. That is, while $Fail(\mathcal{A})$ is not failed, its trajectories basically look like those of $\mathcal{A}$ with the value of the $failed$ variable remaining false throughout the trajectories. The subset for $failed =$ true consists of trajectories of all possible lengths in which all variables are constant. That is, while $Fail(\mathcal{A})$ is failed, its state remains frozen. Note that this does not constrain time from passing, since any constant trajectory, of any length, is allowed.

Performing a failure transformation on the composition $\mathcal{A}\|\mathcal{B}$ of two TIOA results

in a new TIOA whose executions projected to actions and variables of $Fail(\mathcal{A})$ or $Fail(\mathcal{B})$ are in fact executions of $Fail(\mathcal{A})$ or $Fail(\mathcal{B})$ respectively.

3.3 Self-Stabilization of TIOAs

A self-stabilizing system is one that regains normal functionality and behavior sometime after disturbances cease. Here we define self-stabilization for arbitrary TIOAs.

In this section, $A, A_1, A_2, \ldots$ are sets of actions and V is a set of variables. An (A, V)-*sequence* is a (possibly infinite) alternating sequence of actions in $\mathcal{A}$ and trajectories of V. (A, V)-sequences generalize both executions and traces. An (A, V)-sequence is *closed* if it is finite and its final trajectory is closed.

We begin by formally defining what it means for one execution to be a "state-matched" suffix of another:

Definition 3.2. *Given (A, V)-sequences α, α' and $t \geq 0$, α' is a* t-suffix of α *if there exists a closed (A, V)-sequence α'' of duration t such that $\alpha = \alpha''\alpha'$. Execution α' is a* state-matched t-suffix of α *if it is a t-suffix of α, and $\alpha'.fstate$ equals the $\alpha''.lstate$.*

Informally, α' is a state-matched t suffix of α if after t time elapses in α, the system is in the same state as the first state of α', and the remainder of the execution α is equivalent to α'. That is, there exists a closed fragment α'' of duration t, with the same last state as the first state of α' and which when prefixed to α' results in α.

One set S_1 of (A, V)-sequences (say, the sets of executions or traces of some system) stabilizes to another set S_2 (say, desirable behavior) in time t if each state-matched t-suffix of each behavior in set S_1 is included in set S_2. We can think of the set S_1 as the set of executions in which failures, message loss, and other bad phenomena occurs; and we can think of the set S_2 as the set of executions that capture desirable behavior. By saying that S_1 stabilizes to S_2, we are saying that each execution in S_1, after t time, looks just like some execution of S_2.

Definition 3.3. *Given a set S_1 of (A_1, V)-sequences, a set S_2 of (A_2, V)-sequences, and $t \geq 0$, set S_1 is said to* stabilize in time t *to S_2 if each state-matched t-suffix of each sequence in S_1 is in S_2.*

The *stabilizes to* relation is transitive:

Lemma 3.4. *Let S_i be a set of (A_i, V)-sequences, for $i \in \{1, 2, 3\}$. If S_1 stabilizes to S_2 in time t_1, and S_2 stabilizes to S_3 in time t_2, then S_1 stabilizes to S_3 in time $t_1 + t_2$.*

We want to design automata such that if a TIOA starts in any arbitrary state, then eventually it stabilizes to an execution indistinguishable from a correct execution, i.e., eventually it returns to a reachable state. The following definitions help to capture this notion.

First, for any non-empty set L, $L \subseteq Q_{\mathcal{A}}$, we define $Start(\mathcal{A}, L)$ to be the TIOA that is identical to $\mathcal{A}$ except that $\Theta_{Start(\mathcal{A},L)} = L$. That is, its set of start states is L. We define $U(\mathcal{A}) \triangleq Start(\mathcal{A}, Q_{\mathcal{A}})$. Notice that this this new automaton can start in any state. It is straightforward to check that for any TIOA $\mathcal{A}$, the $Fail$ and U operators commute.

We define $R(\mathcal{A}) \triangleq Start(\mathcal{A}, \mathsf{Reach}_{\mathcal{A}})$. That is, $R(\mathcal{A})$ can start in any state that is reachable from a start state of $\mathcal{A}$. Thus any execution of $R(\mathcal{A})$ is an execution fragment of $\mathcal{A}$. A self-stabilizing automaton $\mathcal{A}$ is one where executions of $U(\mathcal{A})$ eventually stabilize to $R(\mathcal{A})$, i.e., to an execution fragment that is reachable from a start state of $\mathcal{A}$.

In fact, we rarely talk about a single automaton running by itself. More often, we deal with a situation where there are multiple automata, composed together to form a single system. Thus, for the purposes of this paper, we define self-stabilization with respect to a system of composed TIOAs. This definition considers the composition of two TIOAs $\mathcal{A}$ and $\mathcal{B}$, allowing $\mathcal{A}$ to start in an arbitrary state while $\mathcal{B}$ starts in a start state. The combination is required to stabilize to a state in a legal set by a certain time.

Definition 3.5. *Let $\mathcal{A}$ and $\mathcal{B}$ be compatible TIOAs, and L be a legal set for the composed TIOA $\mathcal{A}\|\mathcal{B}$. $\mathcal{A}$* self-stabilizes in time t to L relative to $\mathcal{B}$ *if the set of executions of $U(\mathcal{A})\|\mathcal{B}$, that is, $\mathsf{Execs}_{U(\mathcal{A})\|\mathcal{B}}$, stabilizes in time t to executions of $Start(\mathcal{A}\|\mathcal{B}, L)$, that is, to $\mathsf{Execs}_{Start(\mathcal{A}\|\mathcal{B},L)} = \mathsf{Frags}^{L}_{\mathcal{A}\|\mathcal{B}}$.*

4. VIRTUAL STATIONARY AUTOMATA

The Virtual Stationary Automata (VSA) infrastructure has been presented earlier in [Dolev et al. 2005a; Nolte and Lynch 2007a]. The VSA infrastructure can be seen as an abstract system model implemented in middleware, thus providing a simpler and more predictable programming model for the application developer. A VSA layer consists of a set of Virtual Stationary Automata (abstract entities that perform computation) that interact with a set of clients (representing the mobile nodes). This interaction occurs via a virtual broadcast service, which we model with a $VBcast$ automaton (and some additional virtual buffers). For modeling purposes, we also define an automaton that captures the behavior of the real world, and an automaton that captures the behavior of the virtual world. Thus, the main components of the VSA layer are (1) Virtual Stationary Automata (VSA)s, (2) Client Nodes, (3) Real world (RW) and Virtual World (VW) automata, (4) $VBDelay$ buffers, and (5) $VBcast$ broadcast service. The interaction of these components is shown in Figure 1. Each of these components is formally modeled as TIOAs, and the complete system is the composition of the component TIOAs or the corresponding *fail* transformed TIOAs, as the case may be. We now informally describe the architecture of this layer and then briefly sketch its implementation.

4.1 VSA Architecture

For the remainder of this paper, we fix R, the *deployment space*, to be a closed, bounded and connected subset of the plane $\mathbb{R}^2$. The robots all reside in the space defined by R. We fix U to be a totally ordered index set, which we used to identify regions of the plane (as defined in the context of a network tiling). We fix P to be another index set, which we use to identify the participating robots.

Network tiling. A network tiling divides the deployment space R into a set of *regions* $\{R_u\}_{u \in U}$, such that: (i) for each $u \in U$, R_u is a closed, connected subset of R, and (ii) for any $u, v \in U$, R_u and R_v may overlap only at their boundaries. For example,

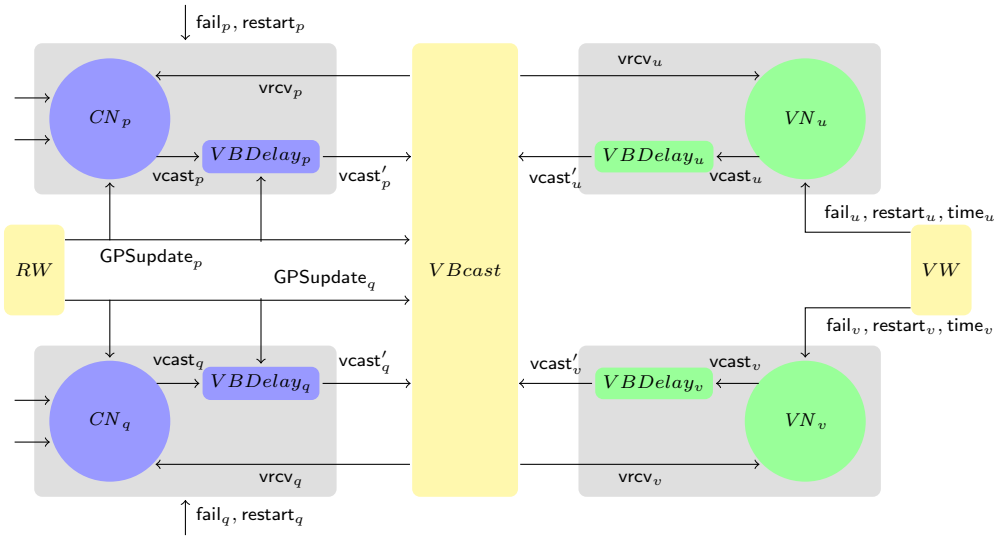

Fig. 1. Virtual Stationary Automata layer.

R might be a large rectangle in the plane, and the network tiling might divide R into squares of edge-length b. We refer to this tiling as the *grid tiling* of R.

For any $u, v \in U$, the region R_u and R_v are said to be *neighbors* if $R_u \cap R_v \neq \emptyset$, i.e., if they shared a boundary. This neighborhood relation $nbrs$ induces a graph on the set of regions where there is an edge between every pair of neighbors. We assume that the network tiling divides R in such a way that the resulting graph is connected. For any $u \in U$, we denote the set of neighboring region identifiers by $nbrs(u)$, and $nbrs^+(u) \triangleq nbrs(u) \cup \{u\}$. We define the distance between two regions u and v, denoted by $regDist(u, v)$, as the number of hops on the shortest path between u and v in the graph. The diameter of the graph, i.e., the distance between the farthest regions in the tiling, is denoted by D, and the largest Euclidean distance between any two points in any region is denoted by r.

We return to our example of a grid tiling where R is divided into $b \times b$ square regions, for some constant $b > 0$. Non-border regions in this tiling have eight neighbors. For a grid tiling with a given b, the diagonal of the tile is of length $\sqrt{2}\, b$, and hence for any two neighboring tiles u and v, the maximum distance between a point in u and a point in v is $2\sqrt{2}\, b$. This implies that r could be any value greater than or equal to $2\sqrt{2}\, b$.

Real World (RW) Automaton. We model the behavior of the real world via the RW automaton. There are two key aspects of the real world: time passes in the real world, and the robots move in the real world. Thus, the RW automaton provides the participating robots with occasional, reliable time and location information, notifying each robot of the current real time and of its current location. (A robot does not learn about the location of other robots, of course.) Such updates happen every so often; in particular, the time between two updates is at most ϵ_{sample}.

Formally, the RW automaton is parameterized by: (a) $v_{max} > 0$, a maximum

speed, and (b) $\epsilon_{sample} > 0$, a maximum time gap between successive updates for each robot. The RW automaton maintains three key variables: (a) A continuous variable now representing true system time; now increases monotonically at the same rate as real-time starting from 0. (b) An array $vel[P \to R \cup \{\perp\}]$; for $p \in P$, $vel(p)$ represents the current velocity of robot p. Initially $vel(p)$ is set to $\perp$, and it is updated by the robots when their velocity changes. (c) An array $loc[P \to R]$; for $p \in P$, $loc(p)$ represents the current location of robot p. Over any interval of time, robot p may move arbitrarily in R provided its path is continuous and its maximum speed is bounded by v_{max}. Automaton RW performs $\mathsf{GPSupdate}(l,t)_p$ actions, $l \in R, t \in \mathbb{R}_{\geq 0}, p \in P$, to inform robot p about its current location and time. For each p, some $\mathsf{GPSupdate}(,)_p$ action must occur every ϵ_{sample} time.

Virtual World (VW) *Automaton.* While the mobile robots live in the real world, the VSAs do not; they reside in a virtual world, and we model the behavior of the virtual world separately from that of the real world. In some ways, this is redundant, as one could define a single entity to model both the real and virtual worlds. It is convenient, however, to model these separately, emphasizing the aspects that are connected to the real world (and the mobile robots), and the aspects that are connected to the virtual world (and the VSAs).

The virtual world automaton VW provides occasional, reliable time information for VSAs. Similar to RW's $\mathsf{GPSupdate}$ action for clients, VW performs $\mathsf{time}(t)_u$ output actions notifying VSA u of the current time. Unlikes the RW, however, it does not provide any location information; the virtual world is a static one, and the VSAs do not move. The time updates occur every so often; one such update occurs at time 0, and they are repeated at least every ϵ_{sample} time thereafter. The VW nondeterministically issues fail_u and $\mathsf{restart}_u$ outputs for each $u \in U$, modeling the fact that VSAs may fail and restart. (Again, notice this is different from the mobile robots and the real world.)

Mobile client nodes. We now discuss how the mobile robots themselves are modeled. For each $p \in P$, the mobile client node CN_p is a TIOA modeling the client-side program executed by the robot with identifier p. CN_p has a local clock variable, *clock* that progresses at the rate of real-time, and is initially $\perp$. CN_p may have arbitrary local variables (albeit none with the name *failed*).

Its external interface includes a $\mathsf{GPSupdate}$ input, to receive updates from the real world. It also includes a facility for sending and receiving messages to/from VSAs. As mentioned previously, we refer to this as the *virtual broadcast service*, and thus each client has an output $\mathsf{vcast}(m)_p$ for sending a message to a VSA, and an input $\mathsf{vrcv}(m)_p$ for receiving a message from a VSA. (This is discussed in more detail below.) A client CN_p may have additional arbitrary other actions (as long as none are name fail or $\mathsf{restart}$). The pseudocode in Figure 2, while a part of our algorithm, at the same time provides an example of how to specify a program for a client node.

As discussed in the previous section, we model the clients, ignoring their behavior when crash failures occur. When defining the behavior of the entire VSA Layer, we use failure transforms to model crash failures.

Virtual Stationary Automata (VSAs). We now discuss how VSAs are modeled. A VSA is a clock-equipped abstract virtual machine. For each $u \in U$, there is a corresponding VSA VN_u which is associated with the geographic region R_u. VN_u has a local clock variable *clock* which progresses at the rate of real-time (it is initially $\perp$). VN_u has the following external interface, which provides it time updates from the VW automaton, and provides it the capacity to send and receive messages via the virtual broadcast service: (a) **Input** $\mathsf{time}(t)_u, t \in \mathbb{R}^{\geq 0}$: models a time update at time t; it sets node VN_u's *clock* to t. (b) **Output** $\mathsf{vcast}(m)_u, m \in Msg$: models VN_u broadcasting message m; (c) **Input** $\mathsf{vrcv}(m)_u, m \in Msg$: models VN_u receiving a message m. VN_u may have additional arbitrary variables (as long as none is named $failed$) and arbitrary internal actions (as long as none is name fail or $\mathsf{restart}$). All such actions must be deterministic.

VBDelay *Automata.* When clients and VSA nodes send messages, there may be some unpredictability in how long it takes for the messages to be delivered. In particular, the messages may be delayed for longer than might be expected due to the costs inherent to emulating the virtual world. We model these delays with a *VBDelay* buffer that delays virtual messages for some additional non-deterministic period of time.

For each client and each VSA node, there is a *VBDelay* buffer that delays messages for up to e time. This buffer intercepts messages just after they are sent. Formally, this implies that the buffer takes as input a $\mathsf{vcast}(m)$ from a node. After some interval of time at most e, the message is handed to the virtual broadcast service. In the case of VSA nodes, the delay $e = 0$, meaning that the message is passed on immediately forwarded to the *VBcast* service with no delay. (It is convenient for treating both clients and VSAs in the same manner to have both attached to VBDelay buffers, even though in the latter case they add no delay.)

VBcast *Automaton.* Finally, we discuss how the virtual broadcast service is modeled. This service is the primary means by which the clients communicate with the VSAs. Each client and VSA has access to the virtual broadcast communication service *VBcast*. The service is parameterized by a constant $d > 0$ which models the upper bound on message delays. *VBcast* takes each $\mathsf{vcast}'(m, f)_i$ input from client and virtual node delay buffers and delivers the message m via $\mathsf{vrcv}(m)$ at each client or virtual node. It delivers the message to every client and VSA that is in the same region as the initial sender, when the message was first sent, along with those in neighboring regions. The *VBcast* service guarantees that in each execution α of *VBcast* there is a correspondence between $\mathsf{vrcv}(m)$ actions and $\mathsf{vcast}'(m, f)_i$ actions, such that: (i) Each vrcv occurs *after and within d time* of the corresponding vcast'. (ii) At most one vrcv at a particular process is mapped to each vcast'. (iii) A message originating from some region u must be received by all robots that are in R_u or its neighbors throughout the transmission period.

Layers and Algorithms.. Since our goal is to model failure-prone robots, we define a *VLayer* to be the composition of the various components described above, where each of the clients has been fail-transformed. That is, each client may fail by crashing.

A VSA layer *algorithm* or a *V-algorithm* is an assignment of a TIOA program to

each client and VSA. (That is, it specified which program execution on each client and VSA.) We denote the set of all V-algorithms as $VAlgs$. Formally:

Definition 4.1. *Let alg be an element of $Valgs$. $VLNodes[alg]$, the* fail-transformed *nodes of the VSA layer parameterized by alg, is the composition of $Fail(alg(i))$ with a* VBDelay *buffer, for all $i \in P \cup U$. $VLayer[alg]$, the* VSA layer parameterized by *alg, is the composition of $VLNodes[alg]$ with $RW \| VW \| VBcast$.*

4.2 VSA Layer Emulation

In [Dolev et al. 2005a; Nolte and Lynch 2007a], we show how mobile nodes can emulate the VSA Layer in a wireless network in which there are no VSAs. That is, we begin with a realistic wireless network in which mobile robots can communicate via wireless broadcast; we then show how to emulate VSAs in such a way as to implement the VSA Layer described in this section. Additional details of this implementation are in [Nolte 2008]. Here we attempt to give some of the basic ideas underlying the implementation. (The remainder of the paper does not depend on the material presented in this section; it is presented only as background information.)

First, the question arises as to under which conditions a VSA Layer can be implemented. In [Nolte 2008], the basic system model is quite similar to the model described in this paper, except that there are no VSAs, and the virtual broadcast service is replaced with a more realistic broadcast service that allows for communication between the mobile nodes. More specifically, the model consists of: (i) mobile nodes, modeled exactly as in this paper, (ii) a RW automaton that models the real world, exactly as in this paper, and (iii) a broadcast service that allows for communication between mobile nodes. The broadcast service guarantees that messages are delivered within some radius r_{real}, and we assume that $r_{real} \geq r + \epsilon_{sample} v_{max}$. This ensures that if a mobile node broadcasts a message, then every other mobile node that is "within range" during the message delivery interval will receive that message. In this case, a mobile node is within range if it is in the same region as the mobile node performing the broadcast, or in a neighboring region. (The second term compensates for the uncertainty in time and location.) The broadcast service guarantees that every message is delivered within time d_{real}. Lastly, the broadcast service satisfies the usual properties, i.e., integrity (a message is delivered only if it was previously sent), and non-duplicative delivery. Given such a broadcast service, Nolte shows how to emulate a VSA Layer satisfying the described properties.

We continue by giving a brief overview of how mobile robots can cooperate to emulate VSAs, thus implementing a VSA Layer. The emulation algorithm is based on a replicated-state-machine paradigm. Specifically, mobile robots in a region R_u cooperate to implement the region u's virtual node.

The emulation relies on a totally ordered broadcast service ($TOBcast$) that guarantees that each mobile robot in a region receives messages in the same order. (This ordered broadcast service is itself implemented via timestamps and additing appropriate timing delays to ensure a uniform delivery sequence.)

All the participants in the emulation protocol for a given VSA act as replicas. Of these replicas, every so often, one is chosen to be the leader. (The leader election service is implemented by a competition among possible candidates.) The leader

is responsible for two tasks. First, it broadcasts the messages that the emulated VSA transmits via vcasts. In this way, the leader helps to emulate the virtual broadcast service. Second, every so often, it broadcasts an up-to-date version of the VSA state. This broadcast is used both to keep the backups synchronized (and hence stabilizing the emulation algorithm), and also to allow new emulators to start participating.

In order to maintain the necessary timing guarantees, the virtual machine state is frozen while these synchronization messages are sent Then, the virtual machine runs at an accelerated pace, simulating the VSN at faster-than-real time until the emulation is caught up.

For further details on the emulation of VSA Layers, we refer the interested reader to [Dolev et al. 2005a; Nolte and Lynch 2007a; Nolte 2008].

5. MOTION COORDINATION USING VIRTUAL NODES

We begin by formally stating the motion coordination problem. We then present an algorithm for the VSA Layer (specifically, a V-algorithm) for solving the motion coordination problem.

5.1 Problem Statement

The goal of motion coordination is to coordinate a set of mobile robots such that they deploy themselves evenly along some curve in the deployment space R. The first step, then, is to define the curve in the plane. Formally, we fix $\Gamma : A \to R$ to be a simple, differentiable curve on R that is parameterized by arc length, where the domain set A of parameter values is an interval in the real line. For example, assuming that the curve is of length at least x, then $\Gamma(x)$ is the point at distance x along Γ.

We also fix a particular network tiling such that each point in Γ is in some region. That is, for the collection of regions $\{R_u\}_{u \in U}$, for each point p in Γ, there is some region R_u such that the point p is in R_u.

For each region, we consider the portion of the curve that intersects that region. Let $A_u \triangleq \{p \in A : region(\Gamma(p)) = u\}$ be the domain of Γ in region u. We assume that A_u is convex for every region u; it may be empty for some u. The local part of the curve Γ in region u is the restriction $\Gamma_u : A_u \to R_u$. We write $|A_u|$ for the length of the curve Γ_u.

In order to more easily discuss the length of the curve, we consider a quantized version of the curve. That is, for some constant σ, we round the length of the curve to the nearest multiple of σ. For example, if the curve is of length 10, and if $\sigma = 3$, then the quantized length of the curve is 12.

More formally, given some quantization constant $\sigma > 0$, we define the *quantization* of a real number x as $q_\sigma(x) = \lceil \frac{x}{\sigma} \rceil \sigma$. We fix σ, and write q_u as an abbreviation for $q_\sigma(|A_u|)$. Notice that, intuitively, q_u is the approximate length of Γ that intersects region R_u, rounded up to the nearest multiple of σ.

We define q_{min} to be the minimum nonzero q_u, i.e., the minimum length of the curve in any region. We define q_{max} as the maximum q_u, i.e., the maximum length of the curve in any region.

Our goal is to design an algorithm for mobile robots such that, once the failures

and recoveries cease, within finite time all the robots are located on Γ and as time progresses they eventually become equally spaced on Γ. Formally, if no fail and restart actions occur after time t_0, then:

(1) There exists a constant T, such that for each $u \in U$, within time t_0+T the set of robots located in R_u becomes fixed and its cardinality is roughly proportional to q_u; moreover, if $q_u \neq 0$ then the robots in R_u are located on[4] Γ_u.

(2) In the limit, as time goes to infinity, all robots in R_u are uniformly spaced[5] on Γ_u.

5.2 Overview of Solution: Motion Coordination Algorithm (MC)

The VSA Layer is used as a means to coordinate the movement of the mobile robots. A VSA controls the motion of the clients in its region by setting and broadcasting target waypoints for the clients. Each VSA VN_u periodically: (i) receives information from clients in its region R_u, (ii) exchanges information with its neighboring VSAs, and (iii) sends out a message containing a calculated target point for each client node "assigned" to region u.

The VSA VN_u performs two tasks when setting the target points: (i) it re-assigns some of the clients that are assigned to itself to neighboring VSAs, and (ii) it sends a target position on Γ to each client that is assigned to itself. The objective of the first task is to spread the mobile robots proportionally among the various regions. By assigning some clients to neighboring regions, a VSA prevents its neighbors from getting depleted of robots. The objective of the second task is to space the nodes uniformly on Γ within each region.

The client algorithm, by contrast, is quite simple. Each client receives a target waypoint from the VSA in its region. It then computes a velocity vector for reaching this target point, and proceeds in this direction as fast as possible.

Of note, each VSA uses only local information about Γ. In particular, its decisions are based only on the (quantized) length of the curve in its region, and in the neighboring regions. For the sake of simplicity, however, we assume that all mobile robots and all VSAs know the complete curve Γ, even though only local information is actually used. (In fact, the mobile robot does not need any information about the curve Γ, as it receives all of its motion planning from VSAs.)

5.3 Client Node Algorithm (CN)

We first describe the algorithm that runs on the mobile robots. The algorithm for the client node CN_p, for a fixed $p \in P$, is presented in Figure 2. The algorithm follows a round structure, where rounds begin at times that are multiples of δ. We refer to the algorithm itself as $CN(\delta)_p$.

At the beginning of each round, each mobile robot sends a cn-update message to the VSA in whose region it is currently residing (see lines 34–38) and stops moving

[4]For a given point $x \in R$, if there exists $p \in A$ such that $\Gamma(p) = x$, then we say that the point x is on the curve Γ; abusing the notation, we write this as $x \in \Gamma$.

[5]A sequence $x_1, \ldots, x_n$ of points in R is said to be *uniformly spaced* on a curve Γ if there exists a sequence of parameter values $p_1 < p_2 \ldots < p_n$, such that for each i, $1 \leq i \leq n$, $\Gamma(p_i) = x_i$, and for each i, $1 < i < n$, $p_i - p_{i-1} = p_{i+1} - p_i$.

(lines 40–45, when $x^* = \perp$). The cn-update message tells the local VSA the robot's *id* and its current location in R.

The local VSA then sends a response to the client, i.e., a target-update message (see lines 47–52). Each such message describes the new target location x_p^* for CN_p, and possibly includes an assignment to a different region. The robot first computes the direction vector toward this new point based on its current position x_p. That is, it computes the (unit-length) direction vector $v_p = (x_p - x_p^*)/||x_p - x_p^*||$. It then proceeds to move in this direction with maximum velocity, i.e., it sets its velocity to $v_{max}v_p$ (see lines 40–45). This is then output as a velocity signal to the RW.

Formally, the *stops when* condition enforces the fact (i) that the robot stops at the beginning of every round, (ii) that it updates its velocity as soon as it receives a target waypoint, and (iii) that it necessarily stops if it does not have enough information to calculate its waypoint. The first situation is resolved by broadcasting a message via a vcast, while the latter two situations are resolved by outputting a new velocity.

Signature:
Input GPSupdate$(l,\ t)_p$, $l \in R$, $t \in \mathbb{R}^{\geq 0}$
Input vrcv$(m)_p$, $m \in$ {target-update} $\times (P \to R)$
Output vcast$(\langle$cn-update, $p,\ l\rangle)_p$, $l \in R$
Output velocity$(vout)_p$, $v \in R^2$

State:
analog *clock*: $\mathbb{R}^{\geq 0} \cup \{\perp\}$, initially $\perp$
analog $x \in R \cup \{\perp\}$, location, initially $\perp$
$x^* \in R \cup \{\perp\}$, target point, initially $\perp$
$v \in \{\perp, 0\} \cup \{v : \mathbb{R}^2 \mid |v| = 1\}$, *initially* $\perp$

Trajectories:
evolve
if $clock \neq \perp$
then $\mathbf{d}(clock) = 1$ **else** $\mathbf{d}(clock) = 0$
if $v \neq \perp$
then $\mathbf{d}(x) = v \cdot v_{max}$ **else** $\mathbf{d}(x) = 0$
stop when
$[x \neq \perp \wedge x^* \neq \perp \wedge clock \bmod \delta = 0 \vee$
$[x \neq \perp \wedge x^* \neq \perp \wedge v||x^* - x|| \neq x^* - x\,] \vee$
$[v \neq 0 \wedge (x = x^* \vee x = \perp \vee x^* = \perp)\,]$

Transitions:
Input GPSupdate$(l,\ t)_p$
Effect
if $\langle x,\ clock\rangle \neq \langle l,\ t\rangle \vee$
$\|x^* - l\| \geq v_{max}(\delta\lceil t/\delta\rceil - t - d_r) \vee$
$x^* = \perp \vee$
$t \mod \delta \notin (e + 2d + 2\epsilon, \delta - d_r)$
then $x, x^* \leftarrow l$; $clock \leftarrow t$
$v \leftarrow \perp$

Output vcast$(\langle$cn-update, $p,\ x\rangle)_p$
Precondition
$x = x \neq \perp \wedge clock \bmod \delta = 0 \wedge x^* \neq \perp$
Effect
$x^* \leftarrow \perp$

Output velocity$(vout)_p$
Precondition
$vout = v_{max} \cdot (x^* - x)/||x^* - x||$
$\vee\, (v = 0 \wedge [x = x^* \vee x^* = \perp \vee x = \perp])$
Effect
$v \leftarrow vout\ /\ v_{max}$

Input vrcv$(\langle$target-update, $target\rangle)_p$
Effect
if $\|target(p) - x\|$
$< v_{max}(\delta\lceil clock/\delta\rceil - clock - d_r)$
and if $clock \bmod \delta > e + 2d + 2\epsilon$
then $x^* \leftarrow target(p)$

Fig. 2. Client node $CN(\delta)_p$ automaton.

5.4 Virtual Stationary Node Algorithm (VN)

We now describe the program for the VSAs. The pseudocode is presented in Figure 3, and the TIOA is parameterized by three parameters: $k \in \mathbb{Z}^+$, and $\rho_1, \rho_2 \in (0, 1)$. The integer k describes the minimum number of robots that should be assigned to a region. We thus refer to k as the *safe* number of robots. When

Signature:
  Input $\mathsf{time}(t)_u$, $t \in \mathbb{R}^{\geq 0}$
  Input $\mathsf{vrcv}(m)_u$,
    $m \in (\{\mathsf{cn\text{-}update}\} \times P \times R)$
      $\cup\ (\{\mathsf{vn\text{-}update}\} \times U \times \mathbb{N})$
  Output $\mathsf{vcast}(m)_u$,
    $m \in (\{\mathsf{vn\text{-}update}\} \times \{u\} \times \mathbb{N})$
      $\cup\ (\{\mathsf{target\text{-}update}\} \times (P \rightharpoonup R))$

State:
  analog *clock*: $\mathbb{R}^{\geq 0} \cup \{\perp\}$, *initially* $\perp$.
  $M : P \rightharpoonup R$, *initially* $\emptyset$.
  $V : U \rightharpoonup \mathbb{N}$, *initially* $\emptyset$.

Trajectories:
  evolve
  if $clock \neq t$
    then $\mathbf{d}(clock) = 1$ **else** $\mathbf{d}(clock) = 0$
  stop when Any precondition is satisfied.

Transitions:
  Input $\mathsf{time}(t)_u$
  Effect
  if $clock \neq t \vee t \bmod \delta \notin (0,\ e + 2d + 2\epsilon]$
    then $M,\ V \leftarrow \emptyset$; $clock \leftarrow t$

  Output $\mathsf{vcast}(\langle \mathsf{vn\text{-}update},\ u,\ n\rangle)_u$
  Precondition
    $clock \bmod \delta = d{+}\epsilon$
    $n = |M| \neq 0 \wedge V \neq \{\langle u, n\rangle\}$
  Effect
    $V \leftarrow \{\langle u,\ n\rangle\}$

  Output $\mathsf{vcast}(\langle \mathsf{target\text{-}update},\ x^*\rangle)_u$
  Precondition
    $clock \bmod \delta = e + 2d + 2\epsilon$
    $M \neq \emptyset$
    $x^* = \mathsf{calctarget}(\mathsf{assign}(M,\ V),\ M)$
  Effect
    $M,\ V \leftarrow \emptyset$

  Input $\mathsf{vrcv}(\langle \mathsf{cn\text{-}update},\ p,\ x\rangle)_u$
  Effect
  if $u = region(x) \wedge clock \bmod \delta \in (0,\ d]$
    then $M(p) \leftarrow x$; $V \leftarrow \emptyset$

  Input $\mathsf{vrcv}(\langle \mathsf{vn\text{-}update},\ p,\ n\rangle)_u$
  Effect
    if $p \in nbrs(u)$ **then** $V(p) \leftarrow n$

Fig. 3. $VN(k, \rho_1, \rho_2)_u$ TIOA, with parameters: safety k, and damping ρ_1, ρ_2.

k is larger, it is less likely that a VSA will fail as all k robots must fail before the VSA fails. When k is smaller, the robots are spread more evenly along the curve (i.e., fewer are "wasted" on regions not on the curve). The parameters ρ_1 and ρ_2 effect the rate of convergence. We refer to the algorithm executing in region R_u as $VN(k, \rho_1, \rho_2)_u$, $u \in U$.

At the beginning of each round, VN_u collects cn-update messages sent from robots located in region R_u (see lines 42–45). It then aggregates the location and round information in a table M. When $d + \epsilon$ time has passed from the beginning of the round, we can be certain that all the cn-update have been delivered. At this point, VN_u computes the number of client nodes that are currently assigned to region R_u, and sends this information in a vn-update message to all of its neighbors (lines 27–32).

When VN_u receives a vn-update message from a neighboring VN, it stores the population information in a table V. When $e+d+\epsilon$ time has elapsed from the point at which it sent its own vn-update passes, we can be sure that VN_u has received vn-update messages from all of its active neighbors. At this point, it calculates how many robots to assign to neighboring regions, and how many to assign to itself. This calculation is performed by the assign function, and the assignments are then used to calculate new target points for local robots via the calctarget function (see Figure 4). These target updates are then broadcast to the client via target-update messages (see lines 34–40).

```
function assign(assignedM : P → R, y : nbrs+(u) → ℕ) =
  assign : P → U, initially {⟨i, u⟩} for each i : assignedM(i) ∈ R_u                    2
  n : ℕ, initially y(u);
  ra : ℕ, initially 0                                                                  4
  if y(u) > k then
    if q_u ≠ 0 then                                                                    6
      let lower = {g ∈ nbrs(u) : (q_g/q_u)y(u) > y(g)}
      for each g ∈ lower                                                               8
        ra ← min(⌊ρ2 · [(q_g/q_u)(y(u) − y(g))]/2(|lower| + 1)⌋, n − k)
        update assign by reassigning ra nodes from u to g                             10
        n ← n − ra
    else if {v ∈ nbrs(u): q_v ≠ 0} = ∅ then                                            12
      let lower = {g ∈ nbrs(u) : y(u) > y(g)}
      for each g ∈ lower                                                               14
        ra ← min(⌊ρ2 · [y(u) − y(g)]/2(|lower| + 1)⌋, n − k)
        update assign by reassigning ra nodes from u to g                              16
        n ← n − ra
    else  ra ← ⌊(y(u) − k)/|{v ∈ nbrs(u) : q_v ≠ 0}|⌋                                   18
      for each g ∈ {v ∈ nbrs(u) : q_v ≠ 0}
        update assign by reassigning ra nodes from u to g                              20
return assign
                                                                                       22
function calctarget(assign: P → U, locM: P → R) =
  seq, indexed list of pairs in A × P, initially the list,                             24
  for each i ∈ P : assign(i)= u ∧ locM(i) ∈ Γ_u, of ⟨p, i⟩
  where p= Γ_u^{-1}(locM(i)),  sorted by p, then i                                     26
  for each i ∈ P : assign(i) ≠ null
    if assign(i) = g ≠ u then locM(i) ← o_g                                            28
    else if locM(i) ∉ Γ_u then locM(i) ← choose {min_{x∈Γ_u}{dist(x, locM(i))}}
    else let p = Γ_u^{-1}(locM(i)), seq(k) = ⟨p, i⟩                                    30
      if k = first(seq) then locM(i) ← Γ_u(inf(A_u))
      else if k = last(seq) then locM(i) ← Γ_u(sup(A_u))                               32
      else let seq(k − 1) = ⟨p_{k−1}, i_{k−1}⟩,
        seq(k + 1) = ⟨p_{k+1}, i_{k+1}⟩                                                34
        locM(i) ← Γ_u(p + ρ1 · (((p_{k−1}+p_{k+1})/2) − p))
return locM                                                                            36
```

Fig. 4. $VN(k, \rho_1, \rho_2)_u$ TIOA functions.

More specifically, the calculation is performed as follows. Consider some specific round, and let $y(u)$ denote the number of robots in region R_u that VSA VN_u reports to its neighbors. (Notice that some of the robots in the region at the beginning of the round may have failed before reporting their presence to VN_u.) Recall that all of u's neighbors receive the value of $y(u)$ via vn-update messages. Also, since every robot knows the location of the curve Γ in its own region, as well as in the neighboring regions, we can assume that the VSA in region R_u knows the value of q_u, as well as q_v for every $v \in nbrs(u)$.

First, if the number of robots assigned to VN_u does not exceed the minimum *safe number* k, then no robots are re-assigned from R_u (see line 5). That is, if $y(u) \leq k$, then there is no change in the assignment.

Next, the change in assignment depends on whether the curve runs through region R_u. If the curve runs through R_u, i.e., if $q_u \neq 0$, then we assign robots based on the length of the curve in R_u and its neighbors (see lines 6–11). Let $lower_u$ denote the subset of $nbrs(u)$ that the curve runs through and have fewer robots than R_u, after normalizing with q_g/q_u. Notice that this normalization factor ensures that the number of robots in each region will be proportional to the length of the curve

running through each region. For each $g \in lower_u$, the VSA VN_u reassigns a number of robots to R_g based on the following. First, we calculate a value ra' that represents the ideal number of robots to transfer:

$$ra' \triangleq \rho_2 \cdot \frac{q_g}{q_u} \cdot \frac{y(u) - y(g)}{2(|lower_u| + 1)} \tag{1}$$

where $\rho_2 < 1$ is a *damping factor*. Then, the VSA VN_u transfers either ra' robots, or the remaining number of robots over k assigned to VN_u. That is, it transfers:

$$ra \triangleq \min(ra', n - k) \tag{2}$$

robots to region R_g. This ensures that at least k robots are left in region R_u.

Next, if the curve does not run the region R_u, i.e., $q_u = 0$, then the transfer of robots depends on whether R_u has any neighbors on the curve. If R_u has no neighbors on the curve, i.e., $q_g = 0$ for all $g \in nbrs(u)$, then the robots are distributed among neighbors that have fewer robots (lines 12–17). Let $lower_u$ denote the subset of $nbrs(u)$ with fewer robots than R_u. In this case, for each $g \in lower_u$, we define ra' as follows:

$$ra' \triangleq \rho_2 \cdot \frac{y(u) - y(g)}{2(|lower_u| + 1)} \tag{3}$$

The VSA VN_u reassigns $ra = \min(ra', n - k)$ robots to R_k.

The last case is when VN_u is on the *boundary* of the curve, meaning that the curve does not run through R_u, but it does run through one of the neighbors of R_u, i.e., there is a $g \in nbrs(u)$ with $q_g \neq 0$ (see lines 18–20). In this case, $y(u) - k$ of VN_u's CNs are assigned equally to neighbors that are on the curve. That is, we calculate ra as follows:

$$ra \triangleq \left\lfloor \frac{y(u) - k}{|\{v \in nbrs(u) : q_v \neq 0\}|} \right\rfloor \tag{4}$$

Again, notice that in each of these cases, at least k robots are left assigned to region R_u.

The calctarget function assigns a target waypoint to every non-failed robot in region R_u. This target point $locM_u(p)$ is in region R_g, where $g = u$ or one of u's neighbors. The target point $locM_u(p)$ is computed as follows: If a robot p is assigned to region R_g, $g \neq u$, then its target is set to the center of region R_g (line 28); if the robot is assigned to R_u but is not located on the curve Γ_u then its target is set to the nearest point on the curve, nondeterministically choosing one if there are several (line 29); if the robot is either the first or last robot on Γ_u then its target is set to the corresponding endpoint of Γ_u (lines 31–32); if the robot is on the curve but is not the first or last client node then its target is moved to the mid-point of the locations of the preceding and succeeding robots on the curve (line 35). For the last two computations a sequence seq of nodes on the curve sorted by curve location is used (line 26).

5.5 Complete System

The complete algorithm, MC, is the instantiation of each component in Figure 1 with fail-transformed CN and VN algorithms. Formally, it is the parallel composition of the following TIOAs: (a) RW, (b) VW, (c) $VBcast$, (d) $Fail(VBDelay_p \| CN_p)$,

one for each $p \in P$, and (e) $Fail(VBDelay_u \| VN_u)$. Recall that $Fail(\mathcal{A})$ denotes the fail-transformed version of TIOA $\mathcal{A}$.

Round length. Given the maximum Euclidean distance, r, between points in neighboring regions, it can take up to r/v_{max} time for a client to reach its target. Also, after the client arrives in the region it was assigned to, it could find the local VN has failed. Let d_r be the time it takes a VN to startup, once a new node enters the region. To ensure a round is long enough for a client node to send the cn-update, allow VNs to exchange information, allow clients to receive a target-update message and arrive at new assigned target locations, and be sure virtual nodes are alive in their region before a new round begins, we require that δ, the CN parameter, satisfy $\delta > 2e + 3d + 2\epsilon + r/v_{max} + d_r$.

6. CORRECTNESS OF ALGORITHM

In this section, we show that *starting from an initial state* the system described in Section 5.2, satisfies the requirements specified in Section 5.1. In the following section we show self-stabilization. The proofs of the results in this section parallel those presented in [Lynch et al. 2005], albeit the semantics of the Virtual Layers used here is different.

We define round t as the interval of time $[\delta(t-1), \delta \cdot t)$. That is, round t begins at time $\delta(t-1)$ and is completed by time $\delta \cdot t$. We say $CN_p, p \in P$, is *active* in round t if node p is not failed throughout round t. A VN_u, $u \in U$, is *active* in round t if there is some active CN_p such that $region(x_p) = u$ for the duration of rounds $t-1$ and t. Thus, by definition, none of the VNs is active in the first round. We also define the following notation:

- $In(t) \subseteq U$ is the subset of VN ids that are is active in round t and $q_u \neq 0$;
- $Out(t) \subseteq U$ is the subset of VNs that are active in round t and $q_u = 0$;
- $C(t) \subseteq P$ is the subset of active CNs at round t;
- $C_{in}(t) \subseteq P$ is the set of active CNs located in regions with id in $In(t)$ at the beginning of round t;
- $C_{out}(t) \subseteq P$ is subset of active CNs located in regions with id in $Out(t)$ at the beginning of round t.

For every pair of regions u, w and for every round t, we define $y(w,t)_u$ to be the value of $V(w)_u$ (i.e., the number of clients u believes are available in region w) immediately prior to VN_u performing a vcast_u in round t, i.e., at time $e + 2d + 2\epsilon$ after the beginning of round t. If there are no new client failures or recoveries in round t, then for every pair of regions $u, w \in nbrs^+(v)$, we can conclude that $y(v,t)_u = y(v,t)_w$, which we denote simply as $y(v,t)$.

We define $\rho_3 \triangleq q_{max}^2/(1-\rho_2)\sigma$. The rate ρ_3 effects the rate of convergence, and will be used in the analysis. Notice that $\rho_3 > 1$.

6.1 Approximately Proportional Distribution

For the rest of this section we fix a particular round number t_0 and assume that, for all $p \in P$, no fail_p or $\mathsf{recover}_p$ events occur at or after round t_0. The first lemma states some basic facts about the assign function.

Lemma 6.1. *In every round $t \geq t_0$: (1) If $y(u,t) \geq k$ for some $u \in U$, then $y(u,t+1) \geq k$; (2) $In(t) \subseteq In(t+1)$; (3) $Out(t) \subseteq Out(t+1)$.*

PROOF. We fix round $t \geq t_0$.

(1) From line 5 of the assign function (Figure 4) it is clear that VN_u, $u \in U$, reassigns some of its CNs in round t. However, it never reassigns more than $n-k$, where n is the number of CNs initially in region R_u. Thus, at least k CNs are not reassigned, and if a CN is not reassigned and does not fail, it remains active in the same region.

(2) For any VN_u, $u \in In(t)$, if $y(u,t) < k$ then VN_u does not reassign CNs, and $y(u,t+1) = y(u,t)$. Otherwise, from line 9 of Figure 4 it follows that $y(u,t+1) \geq k$. In both cases $u \in In(t+1)$.

(3) For any VN_u, $u \in Out(t)$, if $y(u,t) < k$ then VN_u does not reassign CNs, and $y(u,t+1) = y(u,t)$. Otherwise, from line 15 and line 18 of Figure 4 it follows that $y(u,t+1) \geq k$. In both cases $u \in Out(t+1)$.

□

We now identify a round $t_1 \geq t_0$ after which the set of regions $In(t)$ and $Out(t)$ remain fixed.

Lemma 6.2. *There exists a round $t_1 \geq t_0$ such that for every round $t \in [t_1, t_1+(1+\rho_3)m^2n^2]$: (1) $In(t) = In(t_1)$; (2) $Out(t) = Out(t_1)$; (3) $C_{in}(t) \subseteq C_{in}(t+1)$; and (4) $C_{out}(t+1) \subseteq C_{out}(t)$. Round t_1 occurs no later than time $t_0+2m^2\cdot(1+\rho_3)m^2n^2$.*

PROOF. By Lemma 6.1, Part 2, we know that the set $In(t) \subseteq U$ is non-decreasing as t increases. From Part 3, we know that set $Out(t) \subseteq U$ is non-decreasing as t increase. Since U is finite, we conclude from this that there is some round t_1 after which no new regions $u \in U$ are added to either $In(t)$ or $Out(t)$. Thus we have satisfied Parts 1 and 2. Notice that this occurs no later than round $t_0 + 2m^2 \cdot (1+\rho_3)m^2n^2$: in each interval of time $(1+\rho_3)m^2n^2$ either a new region is added to $In(t)$ or $Out(t)$, or the claim is satisfied; there are at most m^2 such regions to add.

For Part 3, consider a client CN_p, $p \in C_{in}(t)$, that is currently assigned in round t to VN_u, $u \in In(t)$. From lines 6–10 of Figure 4 we see that CN_p is assigned to some VN_w, $w \in nbrs^+(u)$ where $q_w \neq 0$. If VN_w is inactive in round $t+1$, then client CN_p remains in VN_w until it becomes active, resulting in VN_w being added to $In(t)$, thus contradicting the fact that for every round $t' \geq t_1$, $In(t') = In(t_1)$. We conclude that VN_w is active in round t, and hence round $t+1$, from which the claim follows.

For Part 4, notice that since there are no failures and recoveries of CNs, $C(t) = C(t+1)$. By definition, $C_{in}(t) \cup C_{out}(t) = C(t)$, $C_{in}(t) \cap C_{out}(t) = \emptyset$, and $C_{in}(t+1) \cup C_{out}(t+1) = C(t+1)$, $C_{in}(t+1) \cap C_{out}(t+1) = \emptyset$. The result follows from Part (3). □

Fix t_1 for the rest of this section such that it satisfies Lemma 6.2. The next lemma states that eventually, regions bordering on the curve stop assigning clients to regions that are on the curve. That is, assume that u is a region where $q_u = 0$, but that u has a neighbor v where $q_v \neq 0$; then, eventually, from some round onwards, u never again assigns clients to v.

Lemma 6.3. *There exists some round $t_2 \in [t_1, t_1 + (1+\rho_3)m^2n^2]$ such that for every round $t \in [t_2, t_2 + (1+\rho_3)m^2n]$: if $u \in Out(t)$ and $v \in In(t)$ and if u and v are neighboring regions, then u does not assign any clients to v in round t.*

Proof. Notice that if u assigns a client to v, then C_{out} decreases by one. During the interval $[t_1, t_1 + (1+\rho_3)m^2n^2]$, we know that C_{out} is non-increasing by Lemma 6.2. Thus, eventually, there is some round t_2 after which either $C_{out} = \emptyset$ or after which no further clients are assigned from a region $Out(\cdot)$ to a region $In(\cdot)$. Since there are at most n clients, we can conclude that this occurs at latest by round $t_1 + n \cdot [(1+\rho_3)m^2n]$. □

Fix t_2 for the rest of this section such that it satisfies Lemma 6.3. Lemma 6.2 implies that in every round $t \geq t_1$, $In(t) = In(t_1)$ and $Out(t) = Out(t_1)$; we denote these simply as In and Out. The next lemma states a key property of the assign function after round t_1. For a round $t \geq t_1$, consider some VN_u, $u \in Out(t)$, and assume that VN_w is the neighbor of VN_u assigned the most clients in round t. Then we can conclude that VN_u is assigned no more clients in round $t+1$ than VN_w is assigned in round t. A similar claim holds for regions in $In(t)$, but in this case with respect to the *density* of clients with respect to the quantized length of the curve. The proof of this lemma is based on careful analysis of the behavior of the assign function.

Lemma 6.4. *In every round $t \in [t_2, t_2+(1+\rho_3)m^2n]$, for $u, v \in U$ and $u \in nbrs(v)$:*

(1) If $u, v \in Out(t)$ and $y(v,t) = \max_{w \in nbrs(u) \cap Out(t)} y(w,t)$ and $y(u,t) < y(v,t)$, then $y(u, t+1) < y(v,t)$.

(2) If $u, v \in In(t)$ and $y(v,t)/q_v = \max_{w \in nbrs(u) \cap In(t)} [y(w,t)/q_w]$ and $y(u,t)/q_u < y(v,t)/q_v$, then:

$$\frac{y(u,t+1)}{q_u} \leq \frac{y(v,t)}{q_v} - (1-\rho_2)\frac{\sigma}{q_{max}^2} \ .$$

Proof. For Part 1, fix u, v and t, as in the statement of the lemma. Consider some region w that is a neighbor of u and that assigns clients to u in round $t+1$. Since $q_u = 0$, notice that w assigns clients to u only if the conditions of lines 12–17 in Figure 4 are met. This implies that $w \in Out(t)$, and hence $y(w,t) \leq y(v,t)$, by assumption. We can also conclude that $lower_w \geq 1$, as w assigns clients to u only if $u \in lower_w$. Finally, from line 15 of Figure 4, we observe that the number of clients that are assigned to u by w in round t is at most:

$$\frac{\rho_2 \left[y(w,t) - y(u,t)\right]}{2(|lower_w(t)| + 1)} \leq \frac{\rho_2 \left[y(v,t) - y(u,t)\right]}{4}$$

Since u has at most four neighbors, we conclude that it is assigned at most $\rho_2 \left[y(v,t) - y(u,t)\right]$ clients. Since $\rho_2 < 1$ and $y(u,t) < y(v,t)$, this implies that:

$$\begin{aligned} y(u,t+1) &\leq y(u,t) + \rho_2 \left[y(v,t) - y(u,t)\right] \\ &\leq \rho_2 \cdot y(v,t) + (1-\rho_2) y(u,t) \\ &< \rho_2 \cdot y(v,t) + (1-\rho_2) y(v,t) \\ &< y(v,t) \ . \end{aligned}$$

For Part 2, as in Part 1, fix u, v and t as in the lemma statement. Recall we have assumed that $y(u,t)/q_u < y(v,t)/q_v$. We begin by showing that, due to the manner in which the curve is quantized, $y(u,t)/q_u \leq y(v,t)/q_v - \sigma/q_{max}^2$. Since q_u is defined as $\lceil P_u/\sigma \rceil \sigma$, and since q_v is defined as $\lceil P_v/\sigma \rceil \sigma$, we notice that, by assumption:

$$y(u,t)\left\lceil \frac{P_v}{\sigma} \right\rceil \sigma < y(v,t)\left\lceil \frac{P_u}{\sigma} \right\rceil$$

We divide both sides by σ, and since both sides are integral, we exchange the '$<$' with a '$\leq$':

$$y(u,t)\left\lceil \frac{P_v}{\sigma} \right\rceil \leq y(v,t)\left\lceil \frac{P_u}{\sigma} \right\rceil - 1$$

From this we conclude:

$$\frac{y(u,t)}{\left\lceil \frac{P_u}{\sigma} \right\rceil} \leq \frac{y(v,t)}{\left\lceil \frac{P_v}{\sigma} \right\rceil} - \frac{\sigma^2}{q_u q_v}$$

Dividing everything by σ, and bounding q_u and q_v by q_{max}, we achieve the desired calculation.

Now, consider some region w that is a neighbor of u and that assigns clients to u in round $t+1$. First, notice that $w \notin Out(t)$, since by Lemma 6.3, no clients are assigned from an *Out* region to an *In* region after round t_2 (prior to $t_2+(1+\rho_3)m^2n$). Thus, w assigns clients to u only if the conditions of lines 6–11 in Figure 4 are met. This implies that $w \in In(t)$, and hence $y(w,t)/q_w \leq y(v,t)/q_v$, by assumption. We can also conclude that $lower_w \geq 1$, as w assigns clients to u only if $u \in lower_w$. Finally, from line 9 of Figure 4, we observe that the number of clients that are assigned to u by w in round t is at most:

$$\frac{\rho_2\left[\left(\frac{q_u}{q_w}\right)y(w,t) - y(u,t)\right]}{2(|lower_w(t)|+1)} \leq \frac{\rho_2\left[\left(\frac{q_u}{q_v}\right)y(v,t) - y(u,t)\right]}{4}$$

Since u has at most four neighbors, we conclude that it is assigned at most $\rho_2\left[(q_u/q_v)y(v,t) - y(u,t)\right]$ clients. This implies that:

$$\begin{aligned} y(u,t+1) &\leq y(u,t) + \rho_2\left[\left(\frac{q_u}{q_v}\right)y(v,t) - y(u,t)\right] \\ &\leq \rho_2\left(\frac{q_u}{q_v}\right)\cdot y(v,t) + (1-\rho_2)\,y(u,t) \end{aligned}$$

Thus, dividing everything by q_u, and recalling that $y(u,t)/q_u \leq y(v,t)/q_v - \sigma/q_{max}^2$:

$$\begin{aligned} \frac{y(u,t+1)}{q_u} &\leq \rho_2\left(\frac{y(v,t)}{q_v}\right) + (1-\rho_2)\cdot\left(\frac{y(u,t)}{q_u}\right) \\ &\leq \rho_2\left(\frac{y(v,t)}{q_v}\right) + (1-\rho_2)\cdot\left(\frac{y(v,t)}{q_v} - \frac{\sigma}{q_{max}^2}\right) \\ &\leq \frac{y(v,t)}{q_v} - (1-\rho_2)\frac{\sigma}{q_{max}^2} \end{aligned}$$

□

The next lemma states that there exists a round T_{out} such that in every round $t \geq T_{out}$, the set of CNs assigned to region $u \in Out(t)$ does not change.

Lemma 6.5. *There exists a round $T_{out} \in [t_2, t_2 + m^2 n$ such that in any round $t \geq T_{out}$, the set of CNs assigned to VN_u, $u \in Out(t)$, is unchanged.*

Proof. First, we show that there exists some round T_{out} such that the aggregate number of CNs assigned to VN_u remains the same in both T_{out} and $T_{out}+1$ for all $u \in Out(t_2)$. We then show that the actual assignment of individual clients remains the same in T_{out} and $T_{out}+1$.

We consider a vector $E(t)$ that represents the distribution of clients among regions in $Out(t)$. That is, the first element in $E(t)$ represents the largest number of clients in any region; the second element in $E(t)$ represents the second largest number of clients in any region; and so forth. We then argue that, compared lexicographically, $E(t+1) \leq E(t)$. Since the elements in $E(t)$ are integers, we conclude from this that eventually the distribution of clients becomes stables and ceases to change.

We proceed to define $E(t)$ as follows for $t \geq t_2$. (We use the notation $\langle \ldots \rangle$ to indicate a vector of elements.) Let $N_{out} = |Out|$. Let $\Pi(t)$ be a permutation of Out that orders the regions by the number of assigned clients, i.e., if u precedes v in $\Pi(t)$, then $y(u,t) \leq y(v,t)$. When we say that some region u has index j, we mean that $\Pi(t)_j = u$. Define $E(t)$ as follows:

$$E(t) = \langle y(\Pi(t)_{N_{out}}, t), y(\Pi(t)_{N_{out}-1}, t), \ldots, y(\Pi(t)_1, t) \rangle \ .$$

We use the notation $E(t)_\ell$ to refer to the ℓ^{th} component of $E(t)$ counting from the right, i.e., it refers to $\Pi(t)_\ell$. Any two vectors $E(t)$ and $E(t+1)$ can be compared lexicographically, examining each of the elements in turn from left to right, i.e., largest to smallest.

We now consider some round $t \in [t_2, t_2 + m^2 n]$, and show that $E(t) \geq E(t+1)$. Consider the case where $E(t) \neq E(t+1)$, and let u be the region with maximum index that assigns clients to another region. Let j be the index of region u.

First, we argue that for every region v with index $\leq j$, we can conclude that $y(v, t+1) < y(u,t)$. Consider some particular region v. Notice that v has no neighbors in Out that are assigned more than $y(u,t)$ clients in round t; otherwise, such a neighbor would assign clients to v, contradicting our choice of u. Thus, by Lemma 6.4, Part 1, we can conclude that $y(v, t+1) < y(u,t)$ (as long as $t \in [t_2, t_2 + 2m^2 n]$, which we will see to be sufficient).

Since this implies that there are at least j regions assigned fewer than $y(u,t) = E(t)_j$ clients in round $t+1$, we can conclude that $E(t+1)_j < E(t)_j$. In order to show that $E(t+1) < E(t)$, it remains to show that for every $j' > j$, $E(t)_{j'} = E(t+1)_{j'}$.

Consider some region v with index $> j$. By our choice of u, it is clear that v is not assigned any clients by a region with index $> j$. It is also easy to see that v is not assigned any clients by a region w with index $\leq j$, since $y(v,t) \geq y(u,t) \geq y(w,t)$; as per line 13, region w does not assign any clients to a region with $\geq y(w,t)$ clients. Thus no new clients are assigned to region v. Moreover, by choice of u, region v assigns none of its clients elsewhere. Finally, since $t \geq t_0$, none of the clients fail. Thus, $y(v,t) = y(v,t+1)$.

Since the preceding logic holds for all $N_{out} - j + 1$ regions with index $> j$, and all have more than $y(u,t) > y(u,t+1)$ clients, we conclude that for every $j' > j$,

$E(t)_{j'} = E(t+1)_{j'}$, implying that $E(t) > E(t+1)$, as desired.

Since $E(\cdot)$ is non-increasing, and since it is bounded from below by the zero vector, we conclude that eventually there is a round T_{out} such that for all $t \geq T_{out}$, $E(t) = E(t+1)$.

Now suppose the set of clients assigned to region u changes in some round $t \geq T_{out}$. The only way the set of clients assigned to region u could change, without changing $y(u,t)$ and the set C_{out}, is if there existed a cyclic sequence of VNs with ids in Out in which each VN gives up $c > 0$ CNs to its successor VN in the sequence, and receives c CNs from its predecessor. However, such a cycle of VNs cannot exist because the *lower* set imposes a strict partial ordering on the VNs.

Finally, we observe that if $E(t) = E(t+1)$ for any t, then the assignment of clients does not change from that point onwards: since all the clients remained in the same regions in $E(t)$ and $E(t+1)$, we can conclude that the *assign* function produced the same assignment in $E(t+1)$ as in $E(t)$. Since the vector $E(\cdot)$ has at most m^2 elements, each with at most n values, we can conclude that T_{out} is at most $m^2 n$ rounds after t_2. □

For the rest of the section we fix T_{out} to be the first round after t_0, at which the property stated by Lemma 6.5 holds. Lemma 6.5, together with Lemmas 6.1, 6.2, and 6.3, imply that in every round $t \geq T_{out}$, $C_{In}(t) = C_{In}(t_1)$ and $C_{Out}(t) = C_{Out}(t_1)$; we denote these simply as C_{In} and C_{Out}. The next lemma states a property similar to that of Lemma 6.5 for VN_u, $u \in In$, and the argument is similar to the proof of Lemma 6.5, and uses Part (2) of Lemma 6.4.

Lemma 6.6. *There exists a round $T_{stab} \in [T_{out}, T_{out} + \rho_3 m^2 n]$ such that in every round $t \geq T_{stab}$, the set of CNs assigned to VN_u, $u \in In$, is unchanged.*

Proof. We proceed to define $E(t)$ as follows for $t \geq T_{out}$. Let $N_{in} = |In|$. Let $\Pi(t)$ be a permutation of In that orders the regions by the density of assigned clients, i.e., if u precedes v in $\Pi(t)$, then $y(u,t)/q_u \leq y(v,t)/q_v$. When we say that some region u has index j, we mean that $\Pi(t)_j = u$. Define $E(t)$ as follows:

$$E(t) = \left\langle \frac{y(\Pi(t)_{N_{in}}, t)}{q_{\Pi(t)_{N_{in}}}}, \frac{y(\Pi(t)_{N_{in}-1}, t)}{q_{\Pi(t)_{N_{in}-1}}}, \ldots, \frac{y(\Pi(t)_1, t)}{q_{\Pi(t)_1}} \right\rangle .$$

We use the notation $E(t)_\ell$ to refer to the ℓ^{th} component of $E(t)$ counting from the right, i.e., it refers to $\Pi(t)_\ell$. Any two vectors $E(t)$ and $E(t+1)$ can be compared lexicographically, examining each of the elements in turn from left to right, i.e., largest to smallest.

We now consider some round $t \geq T_{out}$, and show that $E(t) \geq E(t+1)$. Consider the case where $E(t) \neq E(t+1)$, and let u be the region with maximum index that assigns clients to another region. Let j be the index of region u.

First, we argue that for every region v with index $\leq j$, we can conclude that $y(v,t+1)/q_v \leq y(u,t)/q_u - \zeta$ for some constant ζ. Consider some particular region v. Notice that v has no neighbors in In that have density greater than $y(u,t)/q_u$ in round t; otherwise, such a neighbor would assign clients to v, contradicting our choice of u. Thus, by Lemma 6.4, Part 2, we can conclude that $y(v,t+1)/q_v \leq y(u,t)/q_u - \zeta$ where $\zeta = (1-\rho_2)\frac{\sigma}{q_{max}^2}$ (as long as $t \in [t_2, t_2 + (1+\rho_3)m^2 n]$, which we will see to be sufficient).

Since this implies that there are at least j regions assigned fewer than $y(u,t) = E(t)_j$ clients in round $t+1$, we can conclude that $E(t+1)_j \leq E(t)_j - \zeta$. In order to show that $E(t+1) < E(t)$, it remains to show that for every $j' > j$, $E(t)_{j'} = E(t+1)_{j'}$.

Consider some region v with index $> j$. By our choice of u, it is clear that v is not assigned any clients by a region with index $> j$. It is also easy to see that v is not assigned any clients by a region w with index $\leq j$, since $y(v,t)/q_v \geq y(u,t)/q_u \geq y(w,t)/q_w$; as per line 7, region w does not assign any clients to a region with a density $\geq y(w,t)/q_w$. Thus no new clients are assigned to region v. Moreover, by choice of u, region v assigns none of its clients elsewhere. Finally, since $t \geq t_0$, none of the clients fail. Thus, $y(v,t)/q_v = y(v,t+1)/q_v$.

Since the preceding logic holds for all $N_{in} - j + 1$ regions with index $> j$, and all have more than $y(u,t)/q_u$ clients, we conclude that for every $j' > j$, $E(t)_{j'} = E(t+1)_{j'}$, implying that $E(t) > E(t+1)$, as desired.

Since $E(\cdot)$ is non-increasing, and since it decreases by at least a constant ζ in every round in which it decreases, and since it is bounded from below by the zero vector, we conclude that eventually there is a round T_{stab} such that for all $t \geq T_{stab}$, $E(t) = E(t+1)$.

Now suppose the set of clients assigned to region u changes in some round $t \geq T_{stab}$. The only way the set of clients assigned to region u could change, without changing $y(u,t)/q_u$ and the set C_{in}, is if there existed a cyclic sequence of *VN*s with ids in *In* in which each *VN* gives up $c > 0$ *CN*s to its successor *VN* in the sequence, and receives c *CN*s from its predecessor. However, such a cycle of *VN*s cannot exist because the *lower* set imposes a strict partial ordering on the *VN*s.

Finally, we observe that if $E(t) = E(t+1)$ for any t, then the assignment of clients does not change from that point onwards: since all the clients remained in the same regions in $E(t)$ and $E(t+1)$, we can conclude that the *assign* function produced the same assignment in $E(t+1)$ as in $E(t)$. Since the vector $E(\cdot)$ has at most m^2 elements, each with at most $n\frac{q_{max}^2}{(1-\rho)\sigma}$ values, we can conclude that T_{stab} is at most $\rho_3 m^2 n$ rounds after T_{out}, and hence at most $(1+\rho_3)m^2 n$ rounds after t_2, as needed. □

The following bounds the total number of clients located in regions with ids in *Out* to be $O(m^3)$.

Lemma 6.7. *In every round* $t \geq T_{out}$, $|C_{out}(t)| = O(m^3)$.

Proof. From Lemma 6.5, the set of *CN*s assigned to each VN_u, $u \in Out(t)$, is unchanged in every round $t \geq T_{out}$. This implies that in any round $t \geq T_{out}$, the number of *CN*s assigned by VN_u to any of its neighbors is 0. Therefore, from line 18 of Figure 4, for any boundary VN_v, $(y(v,t) - k)/|In_v| < 1$. Recall that In_v is the (constant) set of neighbors of v with quantized curve length $\neq 0$. Since $|In_v| \leq 4$, $y(v,t) < 4 + k$.

From line 15 of Figure 4, for any non-boundary VN_v, $v \in Out(t)$, if v is 1-hop away from a boundary region u, then $\frac{\rho_2(y(v,t)-y(u,t))}{2(|lower_v(t)|+1)} < 1$. Since $|lower_v(t)| \leq 4$, $y(v,t) \leq \frac{10}{\rho_2} + 4 + k$. Inducting on the number of hops, the maximum number of clients assigned to a VN_v, $v \in Out(t)$, at ℓ hops from the boundary is at most

$\frac{10\ell}{\rho_2} + k + 4$. Since for any ℓ, $1 \leq \ell \leq 2m-1$, there can be at most m VNs at ℓ-hop distance from the boundary, summing gives $|C_{out}| \leq (k+4)(2m-1)m + \frac{10m^2(2m-1)}{\rho_2} = O(m^3)$. □

For the rest of the section we fix T_{stab} to be the first round after T_{out}, at which the property stated by Lemma 6.6 holds. Lemma 6.8 states that the number of clients assigned to each VN_u, $u \in In$, in the stable assignment after T_{stab} is proportional to q_u within a constant additive term. The proof follows by induction on the number of hops from between any pair of VNs.

Lemma 6.8. *In every round $t \geq T_{stab}$, for $u, v \in In(t)$:*

$$\left| \frac{y(u,t)}{q_u} - \frac{y(v,t)}{q_v} \right| \leq \left\lceil \frac{10(2m-1)}{q_{min}\rho_2} \right\rceil.$$

Proof. Consider a pair of VNs for neighboring regions u and v, $u, v \in In$. Assume w.l.o.g. $y(u,t) \geq y(v,t)$. From line 9 of Figure 4, it follows that $\rho_2(\frac{q_v}{q_u} y(u,t) - y(v,t)) \leq 2(|lower_u(t)|+1)$. Since $|lower_u(t)| \leq 4$, $|\frac{y(u,t)}{q_u} - \frac{y(v,t)}{q_v}| \leq \frac{10}{q_v\rho_2} \leq \frac{10}{q_{min}\rho_2}$. By induction on the number of hops from 1 to $2m-1$ between any two VNs, the result follows. □

6.2 Uniform Spacing

From line 29 of Figure 4, it follows that by the beginning of round $T_{stab}+2$, all CNs in C_{in} are located on the curve Γ. Thus, the algorithm satisfies our first goal. The next lemma states that the locations of the CNs in each region u, $u \in In$, are uniformly spaced on Γ_u in the limit, and it is proved by analyzing the behavior of calctarget as a discrete time dynamical system.

Lemma 6.9. *Consider a sequence of rounds $t_1 = T_{stab}, \ldots, t_n$. As $n \to \infty$, the locations of CNs in u, $u \in In$, are uniformly spaced on Γ_u.*

Proof. From Lemma 6.6 we know that the set of CNs assigned to each VN_u, $u \in In$, remains unchanged. Then, at the beginning of round t_2, every CN assigned to VN_u is located in region u and is on the curve Γ_u. Assume w.l.o.g. that VN_u is assigned at least two CNs. Then, at the beginning of round t_3, one CN is positioned at each endpoint of Γ_u, namely at $\Gamma_u(inf(P_u))$ and $\Gamma_u(sup(P_u))$. From lines 31–32 of Figure 4, we see that the target points for these *endpoint* CNs are not changed in successive rounds.

Let $seq_u(t_2) = \langle p_0, i_{(0)}\rangle, \ldots, \langle p_{n+1}, i_{(n+1)}\rangle$, where $y_u = n+2$, $p_0 = inf(P_u)$, and $p_{n+1} = sup(P_u)$. From line 35 of Figure 4, for any i, $1 < i < n$, the i^{th} element in seq_u at round t_j, $j > 2$, is given by:

$$p_i(t_{j+1}) = p_i(t_j) + \rho_1 \left(\frac{p_{i-1}(t_j) + p_{i+1}(t_j)}{2} - p_i(t_j) \right).$$

For the endpoints, $p_i(t_{j+1}) = p_i(t_j)$. Let the i^{th} uniformly spaced point on the curve Γ_u between the two endpoints be x_i. The parameter value $\bar{p}_i$ corresponding to x_i is given by $\bar{p}_i = p_0 + \frac{i}{n+1}(p_{n+1} - p_0)$. In what follows, we show that as $n \to \infty$, the p_i converge to $\bar{p}_i$ for every i, $0 < i < n+1$, that is, the location of the

non-endpoint *CN*s are uniformly spaced on Γ_u. The rest of this proof is exactly the same as the proof of Theorem 3 in [Goldenberg et al. 2004] in which the authors prove convergence of points on a straight line with uniform spacing.

Observe that $\bar{p}_i = \frac{1}{2}(\bar{p}_{i-1} + \bar{p}_{i+1}) = (1-\rho_1)\bar{p}_i + \frac{\rho_1}{2}(\bar{p}_{i-1} + \bar{p}_{i+1})$. Define error at step j, $j > 2$, as $e_i(j) = p_i(t_j) - \bar{p}_i$. Therefore, for each i, $2 \leq i \leq n-1$, $e_i(j+1) = p_i(t_{j+1}) - \bar{p}_i = (1-\rho_1)e_i(j) + \frac{\rho_1}{2}(e_{i-1}(j) + e_{i+1}(j))$, $e_1(j+1) = (1-\rho_1)e_1(j) + \frac{\rho_1}{2}e_2(j)$, and $e_n(j+1) = (1-\rho_1)e_n(j) + \frac{\rho_1}{2}e_{n-1}(j)$. The matrix for this can be written as: $e(j+1) = Te(j)$, where T is an $n \times n$ matrix:

$$\begin{bmatrix} 1-\rho_1 & \rho_1/2 & 0 & 0 & \dots & 0 \\ \rho_1/2 & 1-\rho_1 & \rho_1/2 & 0 & \dots & 0 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & \dots & 0 & \rho_1/2 & 1-\rho_1 & \rho_1/2 \\ 0 & \dots & 0 & 0 & 1-\rho_1 & \rho_1/2 \end{bmatrix}.$$

Using symmetry of T, $\rho_1 \leq 1$, and some standard theorems from control theory, it follows that the largest eigenvalue of T is less than 1. This implies $lim_{j\to\infty} T^j = 0$, which implies $lim_{j\to\infty} e(j) = 0$. □

Thus we conclude by summarizing the results in this section:

Theorem 6.10. *If there are no* fail *or* restart *actions for robots at or after some round t_0, then within a finite number of rounds after t_0:*

(1) *The set of CNs assigned to each VN_u, $u \in U$, becomes fixed, and the size of the set is proportional to the quantized length q_u, within a constant additive term $\frac{10(2m-1)}{q_{min}\rho_2}$.*
(2) *All client nodes in a region $u \in U$ for which $q_u \neq 0$ are located on Γ_u and uniformly spaced on Γ_u in the limit.*

7. SELF-STABILIZATION OF ALGORITHM

In this section we show that the VSA-based motion coordination scheme is self-stabilizing. Specifically, we show that when the VSA and client components in the VSA layer start out in some *arbitrary state* owing to failures and restarts, they eventually return to a reachable state. Thus, the traces of $VLayer[MC]$ running with some reachable state of $Vbcast \| RW \| VW$, eventually, becomes indistinguishable from a reachable trace of $VLayer[MC]$. Recall Definition 4.1 and note that the virtual layer algorithm alg is instantiated here with the motion coordination algorithm MC of Section 5.

We first show that our motion coordination algorithm $VLNodes[MC]$ is self-stabilizing to some set of legal states L_{MC}. Then, we show that these legal states correspond to reachable states of $VLayer[MC]$; hence, the traces of our motion coordination algorithm, where clients and VSAs start in an arbitrary state, eventually look like reachable traces of the correct motion coordination algorithm.

An *emulation* is a kind of implementation relationship between two sets of TIOAs. A VSA layer emulation algorithm is a mapping that takes a VSA layer algorithm, alg, and produces TIOA programs for an underlying system consisting of emulator physical nodes (corresponding to clients), such that when those programs are run with external oracles such as RW, the resulting system has traces that are closely

related to the traces of a VSA layer. In particular, the traces restricted to non-broadcast actions at the client nodes are the same.

In [Dolev et al. 2005a; Nolte and Lynch 2007a] we have shown how to implement a self-stabilizing VSA Layer. In particular, that implementation guarantees that (1) each algorithm $alg \in VAlgs$ stabilizes in some t_{VStab} time to traces of executions of $U(VLNodes[alg])\|R(RW\|VW\|Vbcast)$, and (2) for any $u \in U$, if there exists a client that has been in region u and alive for d_r time and no alive clients in the region failed or left the region in that time, then VSA V_u is not failed. Thus, if the coordination algorithm MC is such that $VLNodes[MC]$ self-stabilizes in some time t to L_{MC} relative to $R(RW\|VW\|Vbcast)$, then we can conclude that physical node traces of the emulation algorithm on MC stabilize in time $t_{VStab} + t$ to client traces of executions of the VSA layer started in legal set L_{MC} and that satisfy the above failure-related properties.

7.1 Legal Sets

First we describe two legal sets for $VLayer[MC]$, L^1_{MC} and L_{MC}. The first legal set L^1_{MC} describes a set of states that result after the first GPSupdate occurs at each client node and the first timer occurs at each virtual node.

Definition 7.1. *A state* $\mathbf{x}$ *of* $VLayer[MC]$ *is in* L^1_{MC} *iff the following hold:*

(1) $\mathbf{x}\lceil X_{Vbcast\|RW\|VW} \in Reach_{Vbcast\|RW\|VW}$.
(2) $\forall u \in U : \neg failed_u : clock_u \in \{RW.now, \bot\} \wedge (M_u \neq \emptyset \Rightarrow clock_u \mod \delta \in (0, e + 2d + 2\epsilon])$.
(3) $\forall p \in P : \neg failed_p \Rightarrow \mathbf{v}_p \in \{RW.vel(p)/v_{max}, \bot\}$.
(4) $\forall p \in P : \neg failed_p \wedge x_p \neq \bot$:
  (a) $x_p = RW.loc(p) \wedge clock_p = RW.now$.
  (b) $x^*_p \in \{x_p, \bot\} \vee ||x^*_p - x_p|| < v_{max}(\delta\lceil clock_p/\delta\rceil - clock_p - d_r)$.
  (c) $Vbcast.reg(p) = region(x_p) \vee clock \mod \delta \in (e + 2d + 2\epsilon, \delta - d_r + \epsilon_{sample})$.

Part (1) requires that $\mathbf{x}$ restricted to the state of $Vbcast\|RW\|VW$ to be a reachable state of $Vbcast\|RW\|VW$. Part (2) states that nonfailed VSAs have *clock*s that are either equal to real-time or $\bot$, and have nonempty M only after the beginning of a round and up to $e + 2d + 2\epsilon$ time into a round. Part (3) states that nonfailed clients have velocity vectors that are equal either to $\bot$ or equal to the client's velocity vector in RW, scaled down by v_{max}. Finally, Part (4) states that nonfailed clients with non-$\bot$ positions have: (4a) positions equal to their actual location and local *clock*s equal to the real-time, (4b) targets that are one of $\bot$, the location, or a point reachable from the current location within d_r before the end of the round, and (4c) *Vbcast* last region updates that match the current region or the time is within a certain time window in a round. It is routine to check that L^1_{MC} is indeed a legal set for *VLayer[MC]*.

Now we describe the main legal set L_{MC} for our algorithm. First we describe a set of *reset* states, states corresponding to states of $VLayer[MC]$ at the start of a round. Then, L_{MC} is defined as the set of states reachable from these reset states.

Definition 7.2. *A state* $\mathbf{x}$ *of* $VLayer[MC]$ *is in* $Reset_{MC}$ *iff:*

(1) $\mathbf{x} \in L^1_{MC}$.

(2) $\forall p \in P : \neg failed_p \Rightarrow$
$[to\_send_p^- = to\_send_p^+ = \lambda \wedge (x_p = \perp \vee \ (x_p^* \neq \perp \wedge v_p = 0))]$.
(3) $\forall u \in U : \neg failed_u \Rightarrow to\_send_u = \lambda$.
(4) $\forall \langle m, u, t, P' \rangle \in vbcastq : P' = \emptyset$.
(5) $RW.now \mod \delta = 0 \wedge \forall p \in P : \forall \langle l, t \rangle \in RW.updates(p) : t < RW.now$.

L_{MC} is the set of reachable states of $Start(VLayer[MC], Reset_{MC})$.

$Reset_{MC}$ consists of states in which (1) in L^1_{MC}, (2) nonfailed clients have empty queues in its $VBDelay$ and either has a position variable equal to $\perp$ or has both a non-$\perp$ target and 0 velocity, (3) nonfailed VSA's have an empty queue in its $VBDelay$, (4) there are no still-processing messages in $Vbcast$, and (5) the time is the starting time for a round and that no GPSupdates have yet occurred at this time. Once again, it is routine to check that that L_{MC} is a legal set for *VLayer[MC]*.

7.2 Stabilization to L_{MC}

First, we state the following result related to stabilization.

Lemma 7.3. *$VLNodes[MC]$ is self-stabilizing to L^1_{MC} in time $t > \epsilon_{sample}$ relative to the automaton $R(Vbcast \| RW \| VW)$.*

To see this, consider the moment after each client has received a GPSupdate and each virtual node has received a time update, which takes at most ϵ_{sample} time.

Next we show that starting from a state in L^1_{MC}, we eventually arrive at a state in $Reset_{MC}$, and hence, a state in L_{MC}.

Lemma 7.4. *Executions of $VLayer[MC]$ started in states in L^1_{MC} stabilize in time $\delta + d + e$ to executions started in states in L_{MC}.*

Proof. It suffices to show that for any length-$\delta + d + e$ prefix α of an execution fragment of $VLayer[MC]$ starting from L^1_{MC}, $\alpha.lstate \in L_{MC}$. By the definition of L_{MC}, it suffices to show that there is at least one state in $Reset_{MC}$ that occurs in α.

Let t_0 be equal to $\alpha.fstate(RW.now)$, the time of the first state in α. We consider all the "bad" messages that are about to be delivered after $\alpha.fstate$. (1) There may be messages in $Vbcast.vbcastq$ that can take up to d time to be dropped or delivered at each process. (2) There may be messages in to_send^- or to_send^+ queues at clients that can submitted to $Vbcast$ and take up to d time to be dropped or delivered at each process. And (3), there may be messages in to_send queues at VSAs that can take up to e time to be submitted to $Vbcast$ and an additional d time to be dropped or delivered at each process. We know that all "bad" messages will be processed (dropped or delivered at each process) by some state $\mathbf{x}$ in α such that $x(RW.now) = t_1 = t_0 + d + e$.

Consider the state $\mathbf{x}^*$ at the start of the first round after state $\mathbf{x}$. Since $\mathbf{x}^*(RW.now) = \delta(\lfloor t_1/\delta \rfloor + 1)$, we have that $\mathbf{x}^*(RW.now) - t_0 = \mathbf{x}^*(RW.now) - t_1 + e + d \leq \delta + e + d$. The only thing remaining to show is that $\mathbf{x}^*$ is in $Reset_{MC}$. It's obvious that $\mathbf{x}^*$ satisfies (1) and (5) of Definition 7.2. Code inspection tells us that for any state in L^1_{MC}, and hence, for any state in α, any new vcast transmissions of messages will fall into one of three categories:

(1) Transmission of cn-update by a client at a time t such that $t \mod \delta = 0$. Such a message is delivered by time $t + d$.
(2) Transmission of vn-update by a virtual node at a time t such that $t \mod \delta = d + \epsilon$. Such a message is delivered by time $t + d + e$.
(3) Transmission of target-update by a virtual node at a time t such that $t \mod \delta = 2d + e + 2\epsilon$. Such a message is delivered by time $t + d + e$.

In each of these cases, any vcast transmission is processed before the start of the next round. Thus, $\mathbf{x}^*$ satisfies properties (2), (3), and (4) of Definition 7.2. To check (2), we just need to verify that for all nonfailed clients if x_p is not $\perp$ then x_p^* is not $\perp$ and v_p is 0. It suffices to show that at least one GPSupdate occurs at each client between state $\mathbf{x}$ and state $\mathbf{x}^*$. (Such an update at a nonfailed client would update x_p^* to be x_p for clients with $x_p^* = \perp$ or x_p^* too far away from x_p to arrive at x_p^* before $\mathbf{x}^*$. Any subsequent receipts of target-update messages will only result in an update to x_p^* if the client will be able to arrive at x_p^* before $\mathbf{x}^*$. This implies that v_p can only be $\perp$ or 0, and since no GPSupdates could have occurred at the same time as $\mathbf{x}^*$, stopping conditions ensure that $v_p \neq \perp$.)

To see that at least one GPSupdate occurs at each client between state $\mathbf{x}'$ and state $\mathbf{x}^*$, we need that $\mathbf{x}^*(RW.now) - \mathbf{x}'(RW.now) > \epsilon_{sample}$. Since $\mathbf{x}^*(RW.now) - \mathbf{x}'(RW.now) = \delta - (\mathbf{x}'(RW.now) \mod \delta) \geq \delta - e - 2d - 2\epsilon$, $\delta > e + 2d + 2\epsilon + d_r$, and $d_r > \epsilon_{sample}$ it follows that $\delta > e + 2d + 2\epsilon + \epsilon_{sample}$. □

Combining our stabilization results we conclude that $VLNodes[MC]$ started in an arbitrary state and run with $R(Vbcast \| RW \| VW)$ stabilizes to L_{MC} in time $\delta + d + e + \epsilon_{sample}$. From transitivity of stabilization and 7.4, the next result follows.

Theorem 7.5. *$VLNodes[MC]$ is self-stabilizing to L_{MC} in time $\delta + d + e + \epsilon_{sample}$ relative to $R(Vbcast \| RW \| VW)$.*

7.3 Relationship between L_{MC} and reachable states

In the previous section we showed that $VLNodes[MC]$ is self-stabilizing to L_{MC} relative to $R(Vbcast \| RW \| VW)$. In order to conclude anything about the traces of $VLayer[MC]$ after stabilization, however, we need to show that traces of $VLayer[MC]$ starting in a state in L_{MC} are reachable traces of $VLayer[MC]$. This is accomplished by first defining a simulation relation $\mathcal{R}_{MC}$ on the states of $VLayer[MC]$, and then proving that for each state $\mathbf{x} \in L_{MC}$, there exists a state $\mathbf{y} \in \mathsf{Reach}_{VLayer[MC]}$ such that $\mathbf{x}$ and $\mathbf{y}$ are related by $\mathcal{R}_{MC}$. This implies that the trace of any execution fragment starting with $\mathbf{x}$ is the trace of an execution fragment starting with $\mathbf{y}$, which is a reachable trace of $VLayer[MC]$. We define the candidate relation $\mathcal{R}_{MC}$ and prove that it is indeed a simulation relation.

Definition 7.6. *$\mathcal{R}_{MC}$ is a relation between states of $VLayer[MC]$ such for any states $\mathbf{x}$ and $\mathbf{y}$ of $VLayer[MC]$, $\mathbf{x} \mathcal{R}_{MC} \mathbf{y}$ iff the following conditions are satisfied:*

(1) $\mathbf{x}(RW.now) = \mathbf{y}(RW.now) \wedge \mathbf{x}(RW.loc) = \mathbf{y}(RW.loc)$.
(2) *For all $p \in P$,* $\mathbf{y}(vel(p)) \in \{\mathbf{x}(vel(p)), \perp\} \wedge$
$\{t \in \mathbb{R}^{\geq 0} \mid \exists l \in R : \langle l, t \rangle \in \mathbf{x}(RW.updates(p))\}$
$= \{t \in \mathbb{R}^{\geq 0} \mid \exists l \in R : \langle l, t \rangle \in \mathbf{y}(RW.updates(p))\}$.

(3) $\mathbf{x}(VW) = \mathbf{y}(VW) \wedge \mathbf{x}(Vbcast.now) = \mathbf{y}(Vbcast.now)$.

(4) $\mathbf{x}(Vbcast.reg) = \mathbf{y}(Vbcast.reg) \wedge$
$\{\langle m, u, t, P'\rangle \in \mathbf{x}(Vbcast.vbcastq) \mid P' \neq \emptyset\}$
$= \{\langle m, u, t, P'\rangle \in \mathbf{y}(Vbcast.vbcastq) \mid P' \neq \emptyset\}$.

(5) *For all* $i \in P \cup U$, $\mathbf{x}(failed_i) = \mathbf{y}(failed_i)$.

(6) *For all* $u \in U : \neg\mathbf{x}(failed_u)$*:*
- (a) $\mathbf{x}(clock_u) = \mathbf{y}(clock_u) \wedge \mathbf{x}(M_u) = \mathbf{y}(M_u)$
$\wedge [\mathbf{x}(M_u) \neq \emptyset \Rightarrow \forall v \in nbrs^+(u) : \mathbf{x}(V_u(v)) = \mathbf{y}(V_u(v))]$.
- (b) $|\mathbf{x}(to\_send_u)| = |\mathbf{y}(to\_send_u)| \wedge \forall i \in [1, |\mathbf{x}(to\_send_u)|] : \forall \langle m, t\rangle = \mathbf{x}(to\_send_u[i]) :$
$\mathbf{y}(to\_send_u[i]) = \langle m, t + \mathbf{y}(rtimer_u) - \mathbf{x}(rtimer_u)\rangle$.

(7) *For all* $p \in P : \neg\mathbf{x}(failed_p)$*:*
- (a) $\mathbf{x}(CN_p) = \mathbf{y}(CN_p) \vee [\mathbf{x}(\mathbf{x}_p) = \mathbf{y}(\mathbf{x}_p) = \perp \wedge \mathbf{x}(v_p) = \mathbf{y}(v_p)]$.
- (b) $\mathbf{x}(VBDelay_p) = \mathbf{y}(VBDelay_p)$.
- (c) $\mathbf{x}(to\_send_p^-) \neq \lambda \Rightarrow \mathbf{x}(Vbcast.oldreg(p)) = \mathbf{y}(Vbcast.oldreg(p))$.

We describe the various conditions two related states $\mathbf{x}$ and $\mathbf{y}$ must satisfy. Part (1) requires that they share the same real-time and locations for CNs. Part (2) requires that for each client, the velocity at RW is equal or the velocity in $\mathbf{y}$ is $\perp$, and GPSupdate records in the two states are for the same times. Part (3) requires that VW's state and $Vbcast.now$ are the same in $\mathbf{x}$ and $\mathbf{y}$. Part (4) requires that the unprocessed message tuples are the same and that the last recorded regions in $Vbcast$ for clients are the same in both states. Part (5) says that failure status of each CN and VN is the same in both states. Part (6a) requires that for a nonfailed VSA, local time and the set M are equal in $\mathbf{x}$ and $\mathbf{y}$, and further, if M is nonempty then V is equal for local regions in both states. Part (6b) says that the to_send queues for a nonfailed VSA are the same, except with the timestamps for messages in $\mathbf{y}$ adjusted up by the difference between $rtimer_u$ in state $\mathbf{y}$ and $\mathbf{x}$. Part (7a) requires that the algorithm state of a nonfailed CN is either the same, or both states share the same local v and have locations equal to $\perp$. Part (7b) says that the $VBDelay$ state is the same for each nonfailed CN in $\mathbf{x}$ and $\mathbf{y}$. Finally, Part (7b) requires that if the $to\_send_p^-$ buffer is nonempty in state $\mathbf{x}$ for a nonfailed client, then $Vbcast.oldreg(p)$ is the same in both states.

The proof of the following lemma is also routine and it breaks-down into a large case analysis. Say that $\mathbf{x}$ and $\mathbf{y}$ are states in $Q_{VLayer[MC]}$ such that $\mathbf{x}\mathcal{R}_{MC}\mathbf{y}$. For any action or closed trajectory σ of $VLayer[MC]$, suppose $\mathbf{x}'$ is the state reached from $\mathbf{x}$, then, we have to show there exists a closed execution fragment β of $VLayer[MC]$ with $\beta.fstate = \mathbf{y}$, $trace(\beta) = trace(\sigma)$, and $\mathbf{x}'\mathcal{R}_{MC}\beta.lstate$.

Lemma 7.7. *$\mathcal{R}_{MC}$ is a simulation relation for $VLayer[MC]$.*

To show that each state in L_{MC} is related to a reachable state of $VLayer[MC]$, it is enough to show that each state in $Reset_{MC}$ is related to a reachable state of $VLayer[MC]$. The proof proceeds by providing a construction of an execution of $VLayer[MC]$ for each state in L_{MC}.

Lemma 7.8. *For each state $\mathbf{x} \in Reset_{MC}$, there exists a state $\mathbf{y} \in Reach_{VLayer[MC]}$ such that $\mathbf{x}\mathcal{R}_{MC}\mathbf{y}$.*

PROOF. Let $\mathbf{x}$ be a state in $Reset_{MC}$. We construct an execution α based on state $\mathbf{x}$ such that $\mathbf{x}\mathcal{R}_{MC}\alpha.lstate$. The construction of α is in three phases. Each phase is constructed by modifying the execution constructed in the prior phase to produce a new valid execution of *VLayer[MC]*. After Phase 1, the final state of the constructed execution shares client locations and real-time values with state $\mathbf{x}$. Phase 2 adds client restarts and velocity actions for nonfailed clients in state $\mathbf{x}$, making the final state of clients consistent with state $\mathbf{x}$. Phase 3 adds VSA restart actions to make the final state of VSAs consistent with state $\mathbf{x}$.

1. Let α_1 be an execution of *VLayer[MC]* where each client and VSA starts out failed, no restart or fail events occur, and $\alpha_1.ltime = \mathbf{x}(RW.now)$. For each failed $p \in P$, there exists some history of movement that never violates a maximum speed of v_{max}, is consistent with stored updates for p, and that lead to the current location of p. We move each failed p in just such a way and add a $\mathsf{GPSupdate}(\langle l, t\rangle)_p$ at time t for each $\langle l, t\rangle \in \mathbf{x}(RW.updates(p))$.
   For each nonfailed $p \in P$ and each state in α_1, we set $RW.loc(p) = \mathbf{x}(RW.loc(p))$ (meaning the client does not move). For each nonfailed $p \in P$, add a $\mathsf{GPSupdate}(\mathbf{x}(RW.loc(p)), t)_p$ action for each t such that $\exists \langle l, t\rangle \in \mathbf{x}(RW.updates(p))$.
   For each $u \in U$, if $\mathbf{x}(last(u)) \neq \perp$ then add a $\mathsf{timer}(t)_u$ output at time t in α_1 for each t in the set $\{t^* \mid t^* = \mathbf{x}(last(u)) \vee (t^* < \mathbf{x}(last(u)) \wedge t^* \mod \epsilon_{sample} = 0)\}$.

   Validity. It is obvious that the resulting execution is a valid execution of *VLayer[MC]*.

   Relation between $\mathbf{x}$ *and* $\alpha_1.lstate$. They satisfy (1)-(4) of Definition 7.6.

2. In order to construct α_2, we modify α_1 in the following way for each $p \in P$ such that $\neg\mathbf{x}(failed_p)$: If $\mathbf{x}(x_p) \neq \perp$, we add a $\mathsf{restart}_p$ event immediately before and a $\mathsf{velocity}(0)_p$ immediately after the last $\mathsf{GPSupdate}_p$ event in α_1. If $\mathbf{x}(x_p) = \perp$ and $\mathbf{x}(v_p) = 0$, then we add a $\mathsf{restart}_p$ and $\mathsf{velocity}(0)_p$ event immediately after the last $\mathsf{GPSupdate}_p$ event in α_1. If $\mathbf{x}(x_p) = \perp$ and $\mathbf{x}(v_p) = \perp$, then we add a $\mathsf{restart}_p$ event at time $\mathbf{x}(RW.now)$ in α_1.

   Validity. Since restart actions are inputs they are always enabled, and a $\mathsf{velocity}_p$ action is always enabled at client CN_p. Also, there can be no trajectory violations since any alive clients receive their first GPSupdate within ϵ_{sample} time of $\mathbf{x}(RW.now)$ in α_2, meaning that since δ is larger than ϵ_{sample} and $\mathbf{x}(RW.now)$ is a round boundary, there is no time before $\mathbf{x}(RW.now)$ in α_2 where a cn-update should have been sent. It is obvious that this is a valid execution of *VLayer[MC]*.

   Relation between $\mathbf{x}$ *and* $\alpha_2.lstate$. They satisfy (1)-(4) and (7) of Definition 7.6.

3. To construct α, we modify α_2 in the following way for each $u \in U$ such that $\neg\mathbf{x}(failed_u)$: If $\mathbf{x}(clock_u) = \perp$, we add a $\mathsf{restart}_u$ event after any time_u actions. If $\mathbf{x}(clock_u) \neq \perp$, we add a $\mathsf{restart}_u$ event immediately before the last time_u action.

   Validity. A restart action is always enabled. Also, there can be no trajectory violations since no outputs at a VSA are enabled until its local M is nonempty. Since M is empty, we can conclude that this is a valid execution of *VLayer[MC]*.

   Relation between $\mathbf{x}$ *and* $\alpha.lstate$. $\mathbf{x}\mathcal{R}_{MC}\alpha.lstate$.

We conclude that α is an execution of *VLayer[MC]* such that if we take $\mathbf{y} = \alpha.lstate$, then $\mathbf{y} \in Reach_{VLayer[MC]}$ and $\mathbf{x}\mathcal{R}_{MC}\mathbf{y}$. □

From Lemmas 7.8 and 7.7 it follows that the set of trace fragments of $VLayer[MC]$ corresponding to execution fragments starting from $Reset_{MC}$ is contained in the set of traces of $R(VLayer[MC])$. Bringing our results together we arrive at the main theorem:

Theorem 7.9. *The traces of $VLNodes[MC]$, starting in an arbitrary state and executed with automaton $R(Vbcast\|RW\|VW)$, stabilize in time $\delta + d + e + \epsilon_{sample}$ to reachable traces of $R(VLayer[MC])$.*

Thus, despite starting from an arbitrary configuration of the VSA and client components in the VSA layer, if there are no failures or restart of client nodes (robots) at or after some round t_0, then within a finite number of rounds after t_0, the clients are located on the curve and are uniformly spaced in the limit.

8. CONCLUSION

We have described how we can use the Virtual Stationary Automaton infrastructure to design protocols that are resilient to failure of participating agents. In particular, we presented a protocol by which the participating robots can be uniformly spaced on an arbitrary curve. The VSA layer implementation and the coordination protocol are both self-stabilizing. Thus, each robot can begin in an arbitrary state, in an arbitrary location in the network, and the distribution of the robots will still converge to the specified curve. The proposed coordination protocol uses only local information, and hence, should adapt well to flocking or tracking problems where the target formation is dynamically changing.

List of Symbols and Functions

δ	Duration of each round of *CN* algorithm, page 16
ϵ_{sample}	Maximum duration between two successive GPS updates, page 11
$\mathsf{Execs}_{\mathcal{A}}$	Set of executions of TIOA $\mathcal{A}$, page 7
$\mathsf{Frags}^{L}_{\mathcal{A}}$	Set of execution fragments of TIOA $\mathcal{A}$ starting from L, page 7
Γ	Target curve; a (fixed) simple differentiable curve on R, page 15
Γ_u	Γ restricted to R_u, page 15
$Fail(\mathcal{A})$	The TIOA obtained by fail-transforming TIOA $\mathcal{A}$, page 8
CN_p	Client node automaton, page 12
RW	Real world automaton, page 12
$VBcast$	Automaton model for virtual layer local broadcast service, page 13
VN_u	Virtual node automaton, page 13
VW	Virtual world automaton, page 12
$\mathsf{Reach}_{\mathcal{A}}$	Reachable states of TIOA $\mathcal{A}$, page 7
$\Theta_{\mathcal{A}}$	Set of start states of TIOA $\mathcal{A}$, page 7
$\mathsf{Traces}_{\mathcal{A}}$	Set of traces of TIOA $\mathcal{A}$, page 7
d	Maximum message delay incurred by *VBcast* service, page 13
e	Maximum delay introduced by *VBDelay* buffer, page 13
$Q_{\mathcal{A}}$	Set of states of TIOA $\mathcal{A}$, page 7

R Deployment space for the mobile robots, page 15

$R(\mathcal{A})$ The TIOA obtained replacing starting states of $\mathcal{A}$ by $\mathsf{Reach}_{\mathcal{A}}$, page 10

R_u A region in R corresponding to VN_u according to some fixed tiling, page 15

$Start(\mathcal{A}, S)$ The TIOA obtained replacing starting states of $\mathcal{A}$ by S, page 10

$U(\mathcal{A})$ The TIOA obtained replacing starting states of $\mathcal{A}$ by $Q_{\mathcal{A}}$, page 10

REFERENCES

Ando, H., Oasa, Y., Suzuki, I., and Yamashita, M. 1999. Distributed memoryless point convergence algorithm for mobile robots with limited visibility. *IEEE Transactions on Robotics and Automation 15,* 5, 818–828.

Blondel, V., Hendrickx, J., Olshevsky, A., and Tsitsiklis, J. 2005. Convergence in multiagent coordination consensus and flocking. In *Proceedings of the Joint forty-fourth IEEE Conference on Decision and Control and European Control Conference*. 2996–3000.

Brown, M. D. 2007. Air traffic control using virtual stationary automata. M.S. thesis, Massachusetts Institute of Technology.

Chandy, K. M., Mitra, S., and Pilotto, C. 2008. Convergence verification: From shared memory to partially synchronous systems. In *In proceedings of Formal Modeling and Analysis of Timed Systems (FORMATS'08)*. LNCS, vol. 5215. Springer Verlag, 217–231.

Chockler, G., Gilbert, S., and Lynch, N. 2008. Virtual infrastructure for collision-prone wireless networks. In *Proceedings of PODC*. To appear.

Clavaski, S., Chaves, M., Day, R., Nag, P., Williams, A., and Zhang, W. 2003. Vehicle networks: achieving regular formation. In *Proceedings of the American control Conference*.

Cortes, J., , Martinez, S., Karatas, T., and Bullo, F. 2004. Coverage control for mobile sensing networks. *IEEE Transactions on Robotics and Automation 20,* 2, 243–255.

Défago, X. and Konagaya, A. 2002. Circle formation for oblivious anonymous mobile robots with no common sense of orientation. In *Proc. 2nd Int'l Workshop on Principles of Mobile Computing (POMC'02)*. ACM, Toulouse, France, 97–104.

Défago, X. and Souissi, S. 2008. Non-uniform circle formation algorithm for oblivious mobile robots with convergence toward uniformity. *Theor. Comput. Sci. 396,* 1-3, 97–112.

Dolev, S. 2000. *Self-stabilization*. MIT Press, Cambridge, MA, USA.

Dolev, S., Gilbert, S., Lahiani, L., Lynch, N., and Nolte, T. 2005a. Virtual stationary automata for mobile networks. In *Proceedings of OPODIS*.

Dolev, S., Gilbert, S., Lahiani, L., Lynch, N. A., and Nolte, T. A. 2005b. Virtual stationary automata for mobile networks. *Technical Report MIT-LCS-TR-979*.

Dolev, S., Gilbert, S., Lynch, N., Shvartsman, A., and Welch, J. 2003. Geoquorums: Implementing atomic memory in ad hoc networks. In *Distributed algorithms*, F. E. Fich, Ed. Lecture Notes in Computer Science, vol. 2848/2003. 306–320.

Dolev, S., Gilbert, S., Lynch, N. A., Schiller, E., Shvartsman, A. A., and Welch, J. L. 2004. Virtual mobile nodes for mobile ad hoc networks. In *18th International Symposium on Distributed Computing (DISC)*. 230–244.

Dolev, S., Gilbert, S., Lynch, N. A., Shvartsman, A. A., and Welch, J. 2005. Geoquorums: Implementing atomic memory in mobile ad hoc networks. *Distributed Computing*.

Efrima, A. and Peleg, D. 2007. Distributed models and algorithms for mobile robot systems. In *SOFSEM (1)*. Lecture Notes in Computer Science, vol. 4362. Springer, Harrachov, Czech Republic, 70–87.

Fax, J. and Murray, R. 2004. Information flow and cooperative control of vehicle formations. *IEEE Transactions on Automatic Control 49*, 1465–1476.

Flocchini, P., Prencipe, G., Santoro, N., and Widmayer, P. 2001. Pattern formation by autonomous robots without chirality. In *SIROCCO*. 147–162.

Gazi, V. and Passino, K. M. 2003. Stability analysis of swarms. *IEEE Transactions on Automatic Control 48,* 4, 692–697.

Goldenberg, D. K., Lin, J., and Morse, A. S. 2004. Towards mobility as a network control primitive. In *MobiHoc '04: Proceedings of the 5th ACM international symposium on Mobile ad hoc networking and computing*. ACM Press, 163–174.

Herman, T. 1996. Self-stabilization bibliography: Access guide. *Theoretical Computer Science*.

Jadbabaie, A., Lin, J., and Morse, A. S. 2003. Coordination of groups of mobile autonomous agents using nearest neighbor rules. *IEEE Transactions on Automatic Control 48,* 6, 988–1001.

Kaynar, D. K., Lynch, N., Segala, R., and Vaandrager, F. 2005. *The Theory of Timed I/O Automata*. Synthesis Lectures on Computer Science. Morgan Claypool. Also available as Technical Report MIT-LCS-TR-917.

Lin, J., Morse, A., and Anderson., B. 2003. Multi-agent rendezvous problem. In *42nd IEEE Conference on Decision and Control*.

Lynch, N., Mitra, S., and Nolte, T. 2005. Motion coordination using virtual nodes. In *Proceedings of 44th IEEE Conference on Decision and Control (CDC05)*. Seville, Spain.

Martinez, S., Cortes, J., and Bullo, F. 2005. On robust rendezvous for mobile autonomous agents. In *IFAC World Congress*. Prague, Czech Republic. To appear.

Nolte, T. and Lynch, N. A. 2007a. Self-stabilization and virtual node layer emulations. In *Proceedings of SSS*. 394–408.

Nolte, T. and Lynch, N. A. 2007b. A virtual node-based tracking algorithm for mobile networks. In *ICDCS*.

Nolte, T. A. 2008. Virtual stationary timed automata for mobile networks. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA.

Olfati-Saber, R., Fax, J., and Murray, R. 2007. Consensus and cooperation in networked multi-agent systems. *Proceedings of the IEEE 95,* 1 (January), 215–233.

Prencipe, G. 2000. Achievable patterns by an even number of autonomous mobile robots. Tech. Rep. TR-00-11. 17.

Prencipe, G. 2001. Corda: Distributed coordination of a set of autonomous mobile robots. In *ERSADS*. 185–190.

Suzuki, I. and Yamashita, M. 1999. Distributed autonomous mobile robots: Formation of geometric patterns. *SIAM Journal of computing 28,* 4, 1347–1363.